

Arrays, Multidimensional arrays, Data Input & Output, Strings and a bit of pointers

Leandro Luciano Gagliardi
lgagliardi@unsam.edu.ar

I/O

Existen 2 tipos de funciones básicas que permiten el movimiento de datos como entrada y como salida:

- **int scanf(const char *format, ...);**
- **int getchar(void);**
- **int printf(const char *format, ...);**
- **int putchar(int c);**

Para utilizarlas debemos incluir la biblioteca **stdio.h** de la siguiente manera:

```
#include <stdio.h>
```

I/O

Las funciones **printf** y **scanf** hacen uso de especificadores de conversión, que especifican (valga la redundancia) el tipo y tamaño del dato. Cada especificador debe comenzar con el signo porcentual (%). Algunos de ellos son los siguientes:

| | | |
|--------|---|--|
| %c | - | print a single character |
| %d, %i | - | print a decimal integer |
| %u | - | print an unsigned decimal integer |
| %o | - | print an unsigned octal integer |
| %x | - | print an unsigned hexadecimal integer using a,b,c,d,e,f |
| %X | - | print an unsigned hexadecimal integer using A,B,C,D,E,F |
| %f | - | print a floating point number(by default 6 digits printed after the point) |
| %e | - | print a floating point number in exponential format. |
| %E | - | same as %e, but it prints E for exponent |
| %g | - | print a floating point number in %f or %e format whichever is shorter |
| %G | - | print a floating point number in %f or %E format whichever is shorter |
| %s | - | print a string |
| %% | - | print a % sign |
| %p | - | print a pointer in hexadecimal format |

getchar() y putchar()

getchar() y putchar() son usados para entrada y salida de caracteres. getchar() es para ingresar un solo caracter y putchar() es para imprimir un solo caracter.

```
#include <stdio.h>

int main(){
    char ch;

    fprintf(stdout, "Enter a character.\n");
    ch = getchar();
    fprintf(stdout, "The entered character is: ");
    putchar(ch);
    fprintf(stdout, "\n");

    return 0;
}
```

Ver `getput.c` en [Clase_2](#)



scanf

```
#include<stdio.h>
int main(void)
{
    int marks;
    .....
    scanf("%d", &marks);
    .....
}
```

```
#include<stdio.h>
int main(void)
{
    char ch;
    .....
    scanf("%c", &ch);
    .....
}
```

```
#include<stdio.h>
int main(void)
{
    char str[30];
    .....
    scanf("%s", str);
    .....
}
```

```
#include<stdio.h>
int main(void)
{
    int basic,da;
    .....
    scanf ("%d%d", &basic, &da);
    .....
}
```

```
#include<stdio.h>
int main(void)
{
    int basic;
    float hra;
    char grade;
    .....
    scanf ("%d%f%c", &basic, &hra, &grade);
    .....
}
```



UNIVERSIDAD
NACIONAL DE
HURLINGHAM



Universidad
Nacional
de San Martín

printf

```
#include<stdio.h>
int main(void)
{
    printf("C is excellent\n");
    return 0;
}
```

```
#include<stdio.h>
int main(void)
{
    int age;
    printf("Enter your age : ");
    scanf("%d",&age);
    return 0;
}
```

```
#include<stdio.h>
int main(void)
{
    int b=1500;
    float h=200.50;
    char g='A';
    printf("Basic=%d, HRA=%f, Grade=%c",b,h,g);
    return 0;
}
```

```
#include<stdio.h>
int main(void)
{
    int b=1500;
    float h=200.50;
    char g='A';
    printf("Basic=%d\tHRA=%f\tGrade=%c\n",b,h,g);
    return 0;
}
```

Formato para enteros

```
#include <stdio.h>

int main(){

    int a, b;
    /* Especificamos que la variable a solo contendrá 2 dígitos */
    fprintf(stdout, "Enter a number to be formated as \"xx\": ");
    scanf("%2d", &a);
    fprintf(stdout, "The number is: %d\n", a);

    /* Especificamos que las variables a y b contendrán 2 y 3 dígitos
     * con un salto de línea en el medio */
    fprintf(stdout, "Enter two numbers to be formated as \"xx\\nxxx\": ");
    scanf("%2d\\n%3d", &a, &b);
    fprintf(stdout, "The numbers are: %d %d\n", a, b);

    /* Especificamos que las variables a y b contendrán 2 y 3 dígitos
     * con un guión en el medio */
    fprintf(stdout, "Enter two numbers to be formated as \"xx-xxx\": ");
    scanf("%2d-%3d", &a, &b);
    fprintf(stdout, "The numbers are: %d-%d\n", a, b);

    return 0;
}
```

Ver [format_int.c](#) en Clase_2



Formato para punto flotante

```
#include <stdio.h>

int main(){
    float c, d;
    /* Especificamos que la variable c solo contendrá 3 dígitos */
    fprintf(stdout, "Enter a float number to be formated as \"xxx\": ");
    scanf("%3f", &c);
    printf("The number is: %3f\n", c);

    /* Especificamos que las variables c y d contendrán 3 y 4 dígitos
     * con un salto de línea en el medio */
    fprintf(stdout, "Enter two float numbers to be formated as \"x.x\\nx.xx\": ");
    scanf("%3f\\n%4f", &c, &d);
    printf("The numbers are: %1.1f %1.2f\\n", c, d);

    /* Especificamos que las variables c y d contendrán 3 y 4 dígitos
     * con un guión en el medio */
    fprintf(stdout, "Enter two float numbers to be formated as \"x.x-xx.x\": ");
    scanf("%3f-%4f", &c, &d);
    printf("The numbers are: %1.1f-%2.1f\\n", c, d);

    return 0;
}
```

Ver `format_float.c` en [Clase_2](#)



Formato para cadenas de caracteres

```
#include <stdio.h>

int main(){

    char str[8];

    /* Formato para strings: */

    fprintf(stdout, "Enter a string: %s\n", str);
    scanf("%3s", str);
    fprintf(stdout, "The entered string is: %s\n", str);

    /* El caracter nulo \0 se agrega al final por defecto */

    fprintf(stdout, "%3s\n", "Hello World!!!\n");
    fprintf(stdout, "%.3s\n", "Hello World!!!\n");
    fprintf(stdout, "%20s\n", "Hello World!!!\n");
    fprintf(stdout, "%8.3s\n", "Hello World!!!\n");

    return 0;
}
```

Ver `format_string.c` en [Clase_2](#)

Compilar estos códigos en casa

1. #define MSSG "Hello World\n"
int main(void)
{
 printf(MSSG);
 return 0;
}

2. int main(void)
{
 printf("Indian\b \n");
 printf("New\rDelhi\n");
 return 0;
}

3. int main(void)
{
 int a=11;
 printf("a=%d\t",a);
 printf("a=%o\t",a);
 printf("a=%x\t",a);
 printf("a=%X\n",a);
 return 0;
}

4. #include<limits.h>
int main(void)
{
 int a=4000000000;
 unsigned int b=4000000000;
 printf("a=%d, b=%u\n",a,b);
 printf("a=%d, b=%u\n",INT_MAX,UINT_MAX);
 return 0;
}



Compilar estos códigos en casa

5. int main(void)
{
 char ch;
 printf("Enter a character:");
 scanf("%c",&ch);
 printf("%d\n",ch);
 return 0;
}

6. int main(void)
{
 float b=123.1265;
 printf("%f\t",b);
 printf("%.2f\t",b);
 printf("%.3f\n",b);
 return 0;
}

7. int main(void)
{
 int a=625,b=2394,c=12345;
 printf("%5d,%5d,%5d\n",a,b,c);
 printf("%3d,%4d,%5d\n",a,b,c);
 return 0;
}

8. int main(void)
{
 int a=98;
 char ch='c';
 printf("%c,%d\n",a,ch);
 return 0;
}

Introducción a punteros

La memoria de una computadora está formada por bytes dispuestos de forma secuencial. Cada byte tiene un número de índice, que se denomina dirección de ese byte. La dirección de estos bytes empieza desde cero y la dirección del último byte es uno menos que el tamaño de la memoria.

Supongamos que tenemos 64 MB de RAM, entonces la memoria constará de:

$$64 \text{ MB} * (10^{10} \text{ kB/MB}) * (10^{10} \text{ B/kB}) = 64 * 1024 * 1024 = 67108864 \text{ B}$$

La dirección de estos bytes será de **0** a **67108863**.

Por eso es necesario declarar una variable antes de utilizarla, ya que el compilador tiene que reservar espacio para ella. El tipo de datos de la variable tiene que ser mencionado para que el compilador sepa cuánto espacio necesita reservarse. Por ejemplo, si declaramos una variable tipo entero (**int var**) el compilador reservará **4 B** en memoria (para un sistema de **64 bits**).

Operador &

Así como cada variable guarda un valor determinado, también debe contar con una posición de memoria, que es el lugar en donde se ubica. Podemos referirnos a esa posición a través de un **puntero**.

C proporciona un operador de dirección **&**, que devuelve la dirección de una variable cuando se coloca antes de ella.

| | |
|---------------------------|---|
| <code>&j;</code> | <code>/*Valid, used with a variable*/</code> |
| <code>&arr[1];</code> | <code>/*Valid, used with an array element */</code> |
| <code>&289;</code> | <code>/*Invalid, used with a constant*/</code> |
| <code>&(j+k);</code> | <code>/*Invalid, used with an expression*/</code> |

Operador &

Ver pointers.c en Clase_2

```
/* Print address of variables using address operator. */
#include<stdio.h>

int main(){
    int age=30;
    float sal = 1500.5;
    printf ("Value of age = %d, Address of age = %u\n", age, &age);
    printf ("Value of sal = %f, Address of sal = %u\n", sal, &sal);

    return 0;
}
```

C_Programming -> ./a.out

Value of age = 30, Address of age = 3024579328

Value of sal = 1500.500000, Address of sal = 3024579332

C_Programming ->

Variable puntero

Al igual que otras variables, los punteros también deben declararse antes de usarse. La sintaxis general de la declaración es:

data_type *pName

Aquí **pName** es el nombre de la variable de puntero. El asterisco * que precede a este nombre informa al compilador que la variable está declarada como un puntero. Aquí el tipo de datos se conoce como el tipo base del puntero. Ejemplo:

```
int *iPtr, age;           //puntero que debe apuntar a variables de tipo int
float *fPtr, sal;         //apunta a variables de tipo float
char *cPtr, ch1, ch2;     //apunta a variables de tipo char
```

Los punteros también son variables, por lo que el compilador reservará espacio para ellos y también tendrán alguna dirección.

Variable puntero

Todos los punteros, independientemente de su tipo base, ocuparán el mismo espacio en la memoria ya que todos ellos contienen solo direcciones.

Por ejemplo, en un sistema de **64 b** las variables puntero ocuparán **8 B**.

Ver pointers.c en Clase_2

```
/* Print address of variables using address operator. */
#include<stdio.h>

int main(){
    int age=30;
    float sal = 1500.5;
    char c = 'A';
    int *pAge = &age;
    float *pSal = &sal;
    char *pChar = &c;

    printf ("Value of age = %d\tAddress of age = %p\n", age, &age);
    printf ("Value of pAge = %p\tAddress of pAge = %p\n", pAge, &pAge);
    printf ("Value of sal = %f\tAddress of sal = %p\n", sal, &sal);
    printf ("Value of pSal = %p\tAddress of pSal = %p\n", pSal, &pSal);
    printf ("Value of char = %c\tAddress of sal = %p\n", c, &c);
    printf ("Value of pChar = %p\tAddress of pChar = %p\n\n", pChar, &pChar);

    printf("sizeof(pAge) = %ld\tsizeof(pSal) = %ld\tsizeof(pChar) = %ld\n",
           sizeof(pAge), sizeof(pSal), sizeof(pChar));

    return 0;
}
```



Variable puntero

```
C_Programming -> ./a.out
Value of age = 30           Address of age = 0x7ffc052b94d8
Value of pAge = 0x7ffc052b94d8  Address of pAge = 0x7ffc052b94e0
Value of sal = 1500.500000    Address of sal = 0x7ffc052b94dc
Value of pSal = 0x7ffc052b94dc  Address of pSal = 0x7ffc052b94e8
Value of char = A           Address of sal = 0x7ffc052b94d7
Value of pChar = 0x7ffc052b94d7 Address of pChar = 0x7ffc052b94f0

sizeof(pAge) = 8           sizeof(pSal) = 8           sizeof(pChar) = 8
```



Puntero nulo

La asignación de **NULL** a un puntero garantiza que no apunte a ninguna ubicación de interfaz válida. Esto se puede hacer de la siguiente manera:

```
ptr = NULL;
```

Operador de indirección

Si colocamos * antes de una variable puntero **pVar** (no en la declaración) entonces podemos acceder a la variable cuya dirección está almacenada en **pVar**. Como **pVar** contiene la dirección de la variable **var**, podemos acceder a la variable a escribiendo ***pVar**.

Ver [pointers_2.c](#) en [Clase_2](#)

```
#include<stdio.h>

int main(){
    int var = 10;
    int *pVar = &var;

    printf ("Value of var = %d\tAddress of var = %p\n", var, &var);
    printf ("Value of *pVar = %d\tAddress of *pVar = %p\n", *pVar, &(*pVar));

    return 0;
}
```

Operador de indirección

```
C_Programming -> ./a.out
Value of var = 10           Address of var = 0x7fff3d6d45ac
Value of *pVar = 10         Address of *pVar = 0x7fff3d6d45ac
```

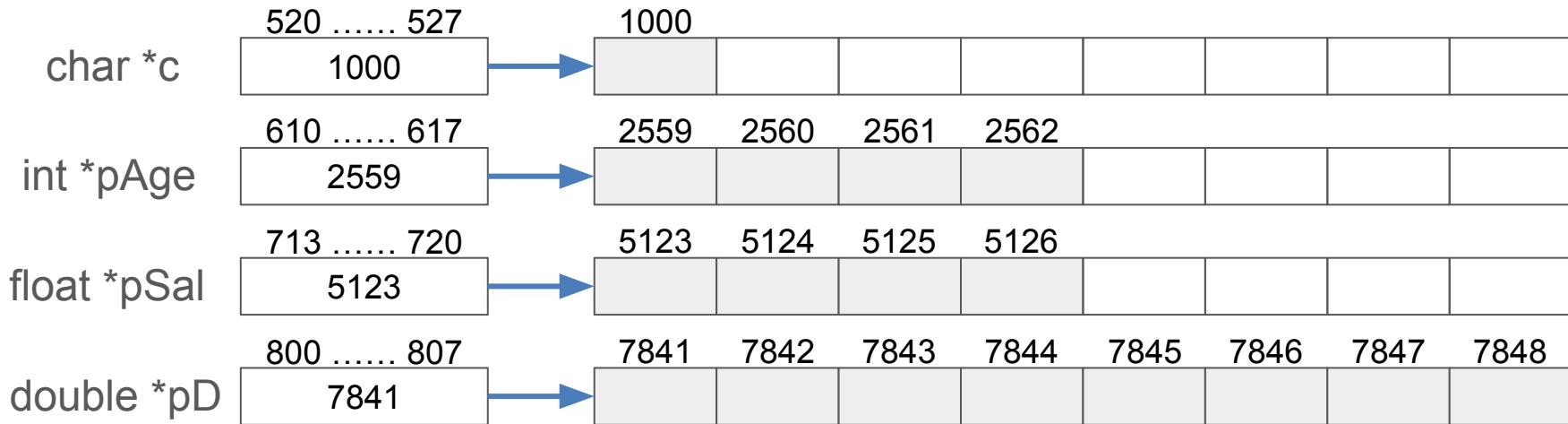
***pVar** es equivalente a **var**
&var es equivalente a **pVar**

El operador de indirección se puede leer como '**valor en la dirección**'. Por ejemplo, ***pVar** se puede leer como '**valor , en la dirección pVar**'.

Este operador de indirección ***** es diferente del asterisco que se utilizó al declarar la variable de puntero.



Operador de indirección



Aritmética de punteros

Las operaciones aritméticas que nunca se pueden realizar con punteros son:

1. Suma, multiplicación, división de dos punteros.
2. Multiplicación entre un puntero y un número cualquiera.
3. División de un puntero por un número cualquiera.
4. Suma de valores float o double a punteros.

Solo se podrán sumar o sustraer números enteros a punteros

Aritmética de punteros

Ver **pointers_3.c** en **Clase_2**

```
#include<stdio.h>

int main(){
    int a = 5;
    int *pA = &a;

    printf ("Value of pA = Address of a = %p\n", pA);
    printf ("Value of ++pA = %p\n", ++pA);
    printf ("Value of pA++ = %p\n", pA++);
    printf ("Value of --pA = %p\n", --pA);
    printf ("Value of pA-- = %p\n", pA--);
    printf ("Value of pA = %p\n", pA);

    return 0;
}
```



Aritmética de punteros

```
C_Programming -> ./a.out
Value of pA = Address of a = 0x7ffdeb704bc
Value of ++pA = 0x7ffdeb704c0
Value of pA++ = 0x7ffdeb704c0
Value of --pA = 0x7ffdeb704c0
Value of pA-- = 0x7ffdeb704c0
Value of pA = 0x7ffdeb704bc
```

Comparación de punteros

Los operadores relacionales `==`, `!=`, `<`, `<=`, `>`, `>=` se pueden utilizar con punteros.

Los operadores `==` y `!=` se utilizan para comparar dos punteros para saber si contienen la misma dirección o no. El uso de estos operadores debe ser del mismo tipo o entre un puntero **NULL** y cualquier otro puntero, o entre punteros **void pointer** y cualquier otro puntero. Los operadores relacionales `<`, `>`; `>=`, `<=` son válidos entre punteros del mismo tipo.

Puntero a puntero

Un puntero es una variable que contiene una dirección de memoria. Esta variable puntero ocupa espacio en la memoria y, por lo tanto, también tiene una dirección, que a su vez puede almacenarse en otra variable. Esto se conoce como variable puntero a puntero. El puntero a puntero se utiliza generalmente al pasar variables puntero a funciones. La sintaxis para declarar un puntero a puntero es la siguiente:

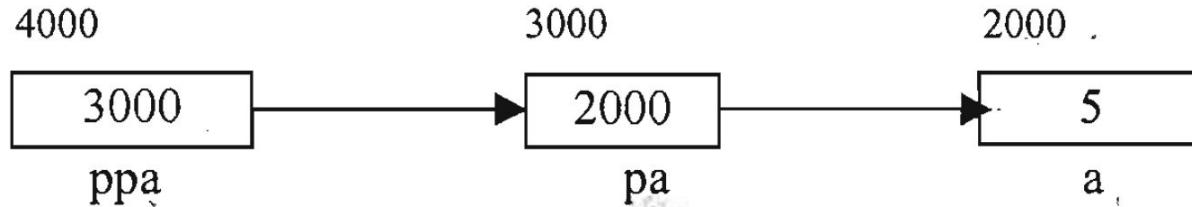
data_type **ptr

Ejemplo:

```
int a;  
int *pA = &a;  
int **ppA = &pA;
```



Puntero a puntero



| Value of a | a | *pa | **ppa | 5 |
|----------------|----|-----|-------|------|
| Address of a | &a | pa | *ppa | 2000 |
| Value of pa | &a | pa | *ppa | 2000 |
| Address of pa | | &pa | ppa | 3000 |
| Value of ppa | | &pa | ppa | 3000 |
| Address of ppa | | | &ppa | 4000 |

Puntero a puntero

Ver pointers_4.c en Clase_2

```
#include <stdio.h>

int main(){
    int a = 5;
    int *pa;
    int **ppa;
    pa = &a;
    ppa=&pa;
    printf("Address of a = %u\n", &a);
    printf("Value of pa = Address of a = %u\n", pa);
    printf("Value of *pa = Value of a = %d\n", *pa);
    printf ("Address of pa = %u\n", &pa);
    printf ("Value of ppa = Address of pa = %u\n" ,ppa);
    printf ("Value of *ppa = Value of pa= %u\n", *ppa);
    printf("Value of **ppa = Value of a = %d\n", **ppa);
    printf("Address of ppa = %u\n", &ppa);

    return 0;
}
```

Puntero a puntero

```
C_Programming -> ./a.out
Address of a = 119237060
Value of pa = Address of a = 119237060
Value of *pa = Value of a = 5
Address of pa = 119237064
Value of ppa = Address of pa = 119237064
Value of *ppa = Value of pa= 119237060
Value of **ppa = Value of a = 5
Address of ppa = 119237072
```

Arrays

Las variables que hemos utilizado hasta ahora pueden almacenar sólo un valor.

El concepto de **array** es útil en situaciones en las que podemos agrupar datos del mismo tipo.

Cada elemento de datos se denomina **elemento del array**. El tipo de datos de los elementos puede ser cualquier tipo de datos válido como **char**, **int** o **float**.

Los elementos del array comparten el mismo **nombre de variable**, pero cada elemento tiene un **subíndice** diferente. En **C** los **subíndices** comienzan desde cero, por lo que **array[0]** es el primer elemento, **array[1]** es el segundo elemento del **array** y así sucesivamente.

Los **arrays** pueden ser **unidimensionales** o **multidimensionales**. La cantidad de **subíndices** determina la dimensión del **array**. Los **arrays** unidimensionales se conocen como **vectores** y los **arrays** bidimensionales se conocen como **matrices**.

Arrays

int arr[n];

| | | | | | | |
|----|----|----|----|---|-----|----|
| 30 | 82 | 53 | 89 | 1 | ... | 71 |
| 0 | 1 | 2 | 3 | 4 | ... | n |

Los elementos de un array están ubicados en direcciones contiguas de la memoria.

int mat[n][m];

| | | | | | |
|-----|-----|-----|-----|-----|----|
| 0 | 1 | 2 | 3 | 4 | m |
| 30 | 82 | 53 | 89 | 1 | 71 |
| 5 | 2 | 12 | 56 | 86 | 99 |
| 45 | 77 | 99 | 22 | 51 | 12 |
| 45 | 34 | 23 | 12 | 8 | 7 |
| 51 | 98 | 56 | 76 | 23 | 3 |
| ⋮ | ⋮ | ⋮ | ⋮ | ⋮ | ⋮ |
| n | 64 | 57 | 12 | 25 | 20 |
| ... | ... | ... | ... | ... | 10 |

Arrays de una dimensión

Para referirnos a un valor en particular del **array** debemos escribir lo siguiente:

arr[n]

Para referirnos a la dirección de un elemento en particular usamos el nombre del array:

arr + n

Donde el nombre **arr** representa la posición de memoria del elemento cuyo subíndice es 0. **n** representa la cantidad de espacios de memoria que nos movemos hacia adelante. Si:

$$\text{arr} = 1000 \text{ y } n = 100 \rightarrow \text{arr} + n = 1100$$

Esto va de la mano con el tema **punteros**.

Arrays de una dimensión

Ver **array_1d.c** en **Clase_2**

```
int size = 15;

/* Declaraciones inválidas */
int intArray2[size] = {0};
size = 17;

/* Declaracion válida */
int intArray[SIZE] = {[0 ... (SIZE/2 - 1)] = 0, [(SIZE/2) ... (SIZE - 1)] = 1};

/* Sin inicializar:
*
*      |__G_|__G_|__G_|__G_|__G_|__G_|__G_|__G_|__G_|__G_|__G_|__G_|__G_|__G_|__G_|__G_|

*/
/* Con la inicialización:
*
*      |__0_|__0_|__0_|__0_|__0_|__0_|__0_|__0_|__1_|__1_|__1_|__1_|__1_|__1_|__1_|__1_|__1_|

*/
*/ */
```



Arrays de una dimensión

```
/* Imprimimos un solo elemento del array */
printf("intArray[4] = %d\n", intArray[4]);

/* Imprimimos intArray */
printf("Initialized intArray values are:\n");
for(int i = 0; i < SIZE; i++)
    printf("intArray[%d] = %d\n", i, intArray[i]);

/* Actualización de los valores de intArray */
for(int i = 0; i < SIZE; i++)
    intArray[i] = 10*i;

/* Imprimimos intArray */
printf("Now, intArray values are:\n");
for(int i = 0; i < SIZE; i++)
    printf("intArray[%d] = %d\n", i, intArray[i]);

return 0;
```

Ver array_1d.c en Clase_2



Arrays de una dimensión

Ver **array_1d_2.c** en **Clase_2**

```
printf("Please, enter values of intaArray:\n");
for(int i = 0; i < SIZE; i++){
    printf("intArray[%d] = ", i);
    scanf("%d\n", intArray + i);
}

/* Imprimimos valores y direcciones de intArray */
printf("Initialized intArray values are:\n");
for(int i = 0; i < SIZE; i++)
    printf("intArray[%d] = %d\t address = %x\n", i, intArray[i], intArray + i);

/* Actualización de los valores de intArray */
for(int i = 0; i < SIZE; i++)
    intArray[i] *= 10;

/* Imprimimos valores y direcciones de intArray */
printf("Now, intArray values are:\n");
for(int i = 0; i < SIZE; i++)
    printf("intArray[%d] = %d\t address = %x\n", i, intArray[i], intArray + i);

return 0;
```

Arrays de dos dimensiones

Para referirnos a un valor en particular de una **matriz** debemos escribir lo siguiente:

mat[n][m]

Para referirnos a la dirección de un elemento en particular usamos el nombre de la **matriz** de la siguiente manera:

mat[n] + m

Donde el nombre **mat[n]** representa la posición de memoria del elemento cuyo subíndice es (0, 0). **m** representa la cantidad de espacios de memoria que nos movemos hacia adelante. Si:

$$\text{mat[n]} = 1000 \text{ y } m = 100 \quad \rightarrow \quad \text{mat[n]} + m = 1100$$

Arrays de dos dimensiones

Ejemplo:

```
int mat[3][4];
```

mat + 0

mat + 1

mat + 2

| | | | | | | | | | | | |
|------------|------------|------------|------------|------------|------------|------------|------------|------------|------------|-------------|-------------|
| mat[2] - 8 | mat[2] - 7 | mat[2] - 6 | mat[2] - 5 | mat[2] - 4 | mat[2] - 3 | mat[2] - 2 | mat[2] - 1 | mat[2] + 0 | mat[2] + 1 | mat[2] + 2 | mat[2] + 3 |
| mat[1] - 4 | mat[1] - 3 | mat[1] - 2 | mat[1] - 1 | mat[1] + 0 | mat[1] + 1 | mat[1] + 2 | mat[1] + 3 | mat[1] + 4 | mat[1] + 6 | mat[1] + 6 | mat[1] + 7 |
| mat[0] + 0 | mat[0] + 1 | mat[0] + 2 | mat[0] + 3 | mat[0] + 4 | mat[0] + 5 | mat[0] + 6 | mat[0] + 7 | mat[0] + 8 | mat[0] + 9 | mat[0] + 10 | mat[0] + 11 |

| | | | | | | | | | | | |
|----|----|----|----|----|----|----|----|----|----|----|----|
| 30 | 82 | 53 | 89 | 30 | 82 | 53 | 89 | 30 | 82 | 53 | 89 |
|----|----|----|----|----|----|----|----|----|----|----|----|

0, 0

0, 1

0, 2

0, 3

1, 0

1, 1

1, 2

1, 3

2, 0

2, 1

2, 2

2, 3

0x010

0x14

0x18

0x1C

0x020

0x24

0x28

0x2C

0x030

0x34

0x38

0x3C

Arrays de dos dimensiones

Ver [array_2d.c en Clase_2](#)

```
/* Inicialización estática */
//int intMatrix[SIZE_R][SIZE_C] = {0};

int intMatrix[SIZE_R][SIZE_C] = {{1, 2, 3, 4},
                                {2, 4, 6, 8},
                                {10, 20, 30, 40}};

/* Inicialización usando memset */
memset(intMatrix, 'a', sizeof(intMatrix));
```

Arrays de dos dimensiones

Ver array_2d.c en Clase_2

```
/* Imprimimos un solo elemento del array */
printf("intMatrix[3][2] = %d\n", intMatrix[2][3]);

/* Imprimimos intArray */
printf("Initialized intMatrix values are:\n");

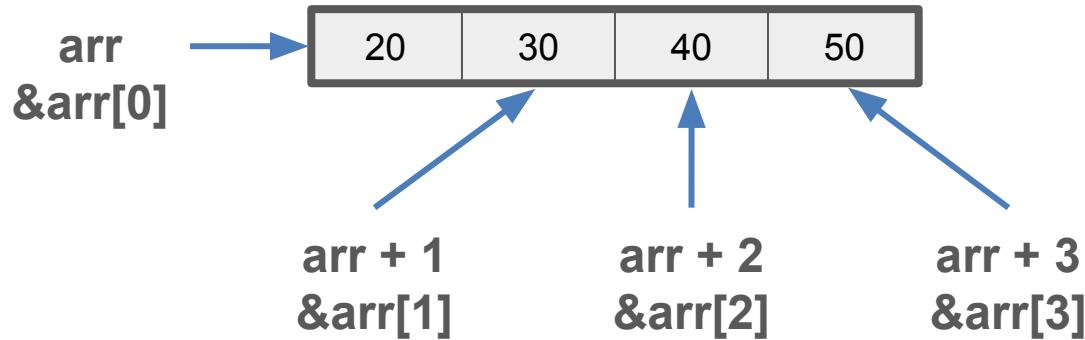
int index = 0;
for(int i = 0; i < SIZE_R; i++)
    for(int j = 0; j < SIZE_C; j++)
        printf("intMatrix[%d][%d] = %c\taddress = %p\n", i, j, intMatrix[i][j], intMatrix[i] + j);

/* Actualización de los valores de intArray */
for(int i = 0; i < SIZE_R; i++)
    for(int j = 0; j < SIZE_C; j++)
        intMatrix[i][j] = 10*i + j;

/* Imprimimos intArray */
printf("Updated intMatrix values are:\n");
for(int i = 0; i < SIZE_R; i++)
    for(int j = 0; j < SIZE_C; j++)
        printf("intMatrix[%d][%d] = %d\taddress = %p\n", i, j, intMatrix[i][j], intMatrix[i] + j);
```

Diferencia entre punteros y nombre de un array

Aplicando aritmética de punteros a un array de nombre **arr** podemos ubicarnos en cualquier posición del mismo sumando un número entero:



Sin embargo, hay diferencias entre una variable de tipo puntero y el nombre de un array.

Diferencia entre punteros y nombre de un array

El nombre de una variable, a diferencia de una variable de tipo puntero, es un puntero constante. Esto significa que no se le puede asignar otra dirección.

```
arr = &num; /* Illegal */  
arr++; /* Illegal */  
arr = arr - 1; /* Illegal */
```

```
ptr = &num; /* Ahora ptr apunta a la variable num */  
ptr++; /* ptr apunta a la siguiente dirección */  
ptr = ptr - 1; /* ptr apunta a la dirección previa */
```

Diferencia entre punteros y nombre de un array

Ver pointers_5.c en Clase_2

```
#include <stdio.h>

int main(){

    int arr[5] = {5, 10, 15, 20, 25};
    int i, *p;
    p = arr;

    for(i = 0; i < 5; i++){
        printf("Address of arr[%d] = %u %u %u %u\n", i, &arr[i], arr + i, p + i, &p[i]);
        printf("Value of arr[%d] = %u %u %u %u\n", i, arr[i], *(arr + i), *(p + i), p[i]);
    }

    return 0;
}
```

Diferencia entre punteros y nombre de un array

```
C_Programming -> ./a.out
Address of arr[0] = 3154917680 3154917680 3154917680 3154917680
Value of arr[0] = 5 5 5 5
Address of arr[1] = 3154917684 3154917684 3154917684 3154917684
Value of arr[1] = 10 10 10 10
Address of arr[2] = 3154917688 3154917688 3154917688 3154917688
Value of arr[2] = 15 15 15 15
Address of arr[3] = 3154917692 3154917692 3154917692 3154917692
Value of arr[3] = 20 20 20 20
Address of arr[4] = 3154917696 3154917696 3154917696 3154917696
Value of arr[4] = 25 25 25 25
```

Strings

Una constante string es una secuencia de caracteres entre comillas dobles. A veces se la denomina literal. Las comillas dobles no forman parte de la cadena.
Ejemplos:

“V”

“Buenos Aires”

“2345”

“” (caracter nulo, contiene solo \0)

“Mi edad es %d y mi altura es %f\n” (string de control en el printf)

Variables String

Una variable de tipo string se declara de la siguiente manera:

```
char str[] = {'B', 'u', 'e', 'n', 'o', 's', ' ', 'A', 'i', 'r', 'e', 's', '\0'};  
char str[] = "Buenos Aires";
```

Esta variable se almacenará en memoria como un array de caracteres:

| 1000 | 1001 | 1002 | 1003 | 1004 | 1005 | 1006 | 1007 | 1008 | 1009 | 1010 | 1011 | 1012 |
|--------|--------|--------|--------|--------|--------|--------|--------|--------|--------|---------|---------|---------|
| str[0] | str[1] | str[2] | str[3] | str[4] | str[5] | str[6] | str[7] | str[8] | str[9] | str[10] | str[11] | str[12] |
| 'B' | 'u' | 'e' | 'n' | 'o' | 's' | ' ' | 'A' | 'i' | 'r' | 'e' | 's' | '\0' |

Variables String

Ver string4.c en Clase_2

```
#include <stdio.h>

int main(){

    char str[] = "Argentina";
    char *p;
    p = str;
    while(*p != '\0'){
        printf("Character = %c\t" , *p) ;
        printf("Address = %p\n" , p) ;
        p++;
    }

    return 0;
}
```

Variables String

```
C_Programming -> ./a.out
Character = A    Address = 0x7ffd478456ce
Character = r    Address = 0x7ffd478456cf
Character = g    Address = 0x7ffd478456d0
Character = e    Address = 0x7ffd478456d1
Character = n    Address = 0x7ffd478456d2
Character = t    Address = 0x7ffd478456d3
Character = i    Address = 0x7ffd478456d4
Character = n    Address = 0x7ffd478456d5
Character = a    Address = 0x7ffd478456d6
```

gets() y puts()

gets() y **puts()** son usados para entrada y salida de cadenas de caracteres. **getc()** es para ingresar una cadena y **putc()** es para imprimir una cadena.

```
#include <stdio.h>

int main(){
    char name[20];
    printf("Enter name: ");
    gets(name);
    printf("Entered name: ");
    puts(name);

    return 0;
}
```

```
C_Programming -> ./a.out
Enter name: Leandro
Entered name: Leandro
```



Ver **string5.c** en **Clase_2**

Otras funciones para el manejo de strings

Usar el comando `$ man` para descubrir el uso correcto de las siguientes funciones:

- `strlen()`
- `strcmp()`
- `strcpy()`
- `strcat()`
- `sprintf()`
- `sscanf()`

```
#include <stdio.h>
#include <string.h>

int main(){
    char name[30] = "String de Prueba.";
    char name2[30] = {'\0'};
    char name3[30] = {'\0'};
    char number[] = "1000";
    int num = 0;

    /* Longitud de strings */
    printf("La longitud de name es: %d\n", strlen((const char *)name));
    printf("name2 es inicializado a: %s\n", name2);
    printf("name3 es inicializado a: %s\n", name3);
    /* Se puede usar también puts(name2); */

    /* Copia de strings */
    strcpy(name2, name);
    printf("name2 ahora vale: %s\n", name2);

    /* Concatenación de strings */
    strcat(name, " -> Concatenamos aquí.\n");

    /* sprintf es como printf pero en lugar de enviar la salida
     * formateada a stdout, lo guarda en una variable string. */
    sprintf(name3, "Número de Prueba: %d\n", 123U);

    /* fprintf guarda la salida formateada en un archivo.
     * En este caso particular a la salida estándar stdout. */
    fprintf(stdout, "%s", name3);

    /* sscanf hace lo mismo que scanf pero lee el contenido de
     * un string y no de la entrada estándar stdin, y
     * convierte lo leído a variables de diferentes tipos. */
    printf("El valor de num es = %d\n", num);
    sscanf(number, "%d", &num);
    printf("El valor de num ahora es = %d\n", num);

    return 0;
}
```

strlen() no considera el carácter nulo.

```
C_Programming -> ./a.out
La longitud de name es: 17
name2 es inicializado a:
name3 es inicializado a:
name2 ahora vale: String de Prueba.
Número de Prueba: 123
El valor de num es = 0
El valor de num ahora es = 1000
C_Programming -> vim string6.c
```

Ver [string6.c](#) en [Clase_2](#)



strcmp() vs ==

Ver string3.c en Clase_2

```
#include <stdio.h>
#include <string.h>

int main(){

    char c[] = "bad";
    char d[] = "bad";
    printf("%u\n", "good");
    printf("%u\n", "good");

    /* Acá estamos comparando cadenas de caracteres constantes. */
    if("bad" == "bad")
        printf("Same\n");
    else
        printf("Not same\n");

    /* Acá estamos comparando direcciones de memoria */
    if(c == d)
        printf("Same\n");
    else
        printf("Not same\n");

    /* Acá estamos comparando el contenido de dos variables string */
    if (!strcmp(c, d))
        printf("Same\n");
    else
        printf("Not same\n");

    return 0;
}
```

```
C_Programming -> ./a.out
2193715204
2193715204
Same
Not same
Same
```



Extra Slides

Leandro Gagliardi
lgagliardi@unsam.edu.ar

Arrays con más de dos dimensiones

Para referirnos a un valor en particular de un **array** de 3 dimensiones debemos escribir lo siguiente:

arr3[n][m][o]

Para referirnos a la dirección de un elemento en particular usamos el nombre de la **matriz** de la siguiente manera:

arr3[n][m] + o

Donde el nombre **arr3[n][m]** representa la posición de memoria del elemento cuyo subíndice es (0, 0, 0).

m representa la cantidad de espacios de memoria que nos movemos hacia adelante. Si:

`arr3[n][m] = 1000 y o = 100 -> arr3[n][m] + o = 1100`