Informática

Unidad 2D, 2E y 2F

Ingeniería en Mecatrónica

Facultad de Ingeniería
Universidad Nacional de Cuyo

Dr. Ing. Martín G. Marchetta
martin.marchetta@ingenieria.uncuyo.edu.ar

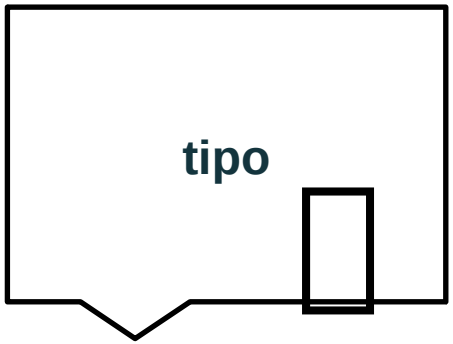
Ing. Sebastián C Cardello
sebastian.cardello@ingenieria.uncuyo.edu.ar

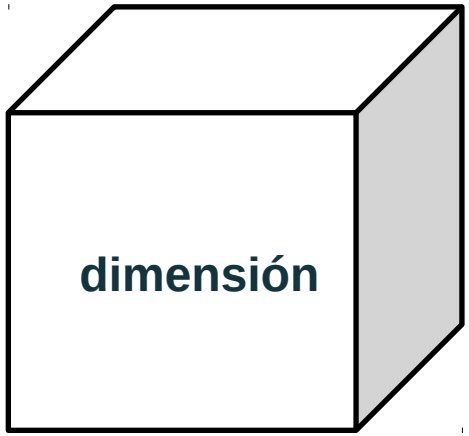
UNIVERSIDAD NACIONAL DE CUYO

FACULTAD DE INGENIERIA
en acción continua...

2D – Tipos de datos, Operadores, Expresiones y Sentencias

Datos y su clasificación





Tipos numéricos

Enteros

- Diferentes tamaños:
  - short (entero corto, 2 bytes)
  - int (entero, 4 bytes)
  - long (entero doble, 8 bytes)
- Con y sin signo (unsigned).
- Los valores constantes se escriben sin separador de miles.
- Ej.:

```
int a = -3782;

unsigned long s = 16513;
```

Reales

- Diferentes precisiones: de 4 a 8 bytes (en general).
  - float (decimal de simple precisión, 4 bytes)
  - double (decimal de doble precisión, 8 bytes)
- Se representan mediante un mecanismo de coma flotante (símil notación científica).
- El punto separa decimales.
- Ej.:

```
float a = 13.18;

double a = -4e-9;
```

Tipo lógico (booleano)

- Admite 2 valores: verdadero (true) o falso (false)
- De vital importancia para las operaciones lógicas/relacionales.
- C no dispone de un tipo específico para los booleanos, pero se pueden definir como constantes
- Ej.

```
#define TRUE  1
#define FALSE 0
```

Tipo caracter y cadenas

- Representan textos.
- Se encierran entre comillas (dobles o simples)
  - un caracter (unidimensional): ‘8’
  - una cadena (multidimensional): “esto es una cadena”
- En C el carácter se representa con el tipo `char`.
  - Representado como un numérico de 1 byte.
  - Las cadenas son vectores de caracteres.
  - Las cadenas poseen un carácter especial ‘\0’ que identifica el fin de la misma
- Ej.

```
char letra=’a’;
char hola[] = “HOLA”;
char otro_hola[]={'H','o','l','a','\0'}; // Igual al anterior
```

Variables

- Valores mutables.
- Exigen una identificación válida.
  - Deben comenzar con una letra
  - No pueden usarse palabras reservadas
- Definición:

```
<tipo_de_dato> <nombre_variable> [=<expresión de inicialización>];
```

- ¿Definiciones correctas o incorrectas?

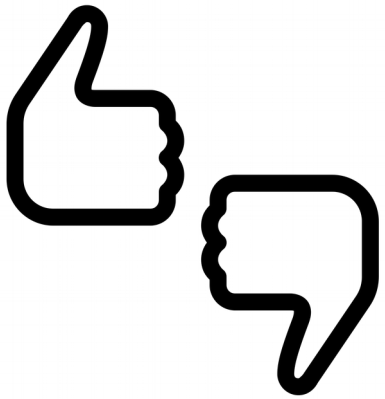
A) `int edad;`

B) `num = 15;`

C) `char int = ‘a’;`

D) `double ¥ = 1000;`

E) `char 8000[] = “ocho mil”;`



Constantes

- Valores inmutables
- Opcionalmente identificables
- En C existen dos formas
  - A través de una macro del preprocesador `#define`
  - A través del calificativo `const`
- Ej.

```
#define MAX_LINEAS = 100
const float PI = 3.1415;
```

Declarar vs asignar

- Declarar es identificar y tipificar una variable para un programa
- La declaración implica una reserva de memoria
- Asignar es dar valor a una variable o constante
- Una vez declarada sólo pueden ser asignados valores de su tipo (o uno compatible)
- La primera asignación de una variable o constante se denomina inicializar
- Una constante nombrada sólo puede ser inicializada

Expresiones

±	≤	&	C
Aritméticas	Relacionales	Lógicas	Cadenas

Aritméticas

+ (suma y unitario positivo)

- (resta y unitario negativo)

* (multiplicación)

/ (división)

// o **floor()** (división entera)

** o **pow()** (exponenciación)

% (módulo o resto)

++ -- (incremento y decremento)

operandos:
tipo numérico
retornan:
tipo numérico

```
ej.
resultado*8.5
10%2*5
sueldo++
```

Relacionales

< (menor que)

> (mayor que)

== (igual que)

!= (distinto de)

<= (menor o igual que)

>= (mayor o igual que)

operandos:
tipos evaluables
retornan:
tipo lógico

ej.
`A * 2 == 16`
`‘1’ != 36`
`‘a’ > 40`

Lógicos y a nivel de bits

Lógicos

- **&& (conjunción)**
- **|| (disyunción)**
- **! (negación)**

a	*b*	*!a*	*a && b*	*a || b*
0	0	1	0	0
0	1	1	0	1
1	0	0	0	1
1	1	0	1	1

Bit

- &
- |
- ~ (invierte los bits)
- ^ (1 si los operando son distintos)
- >> (corrimiento de bits a la derecha)
- << (corrimiento a la izquierda)

operando:
tipo lógico / bit
retornan:
tipo lógico / bit

ej.

```
a > 0 && b > 0
! val || c == ‘a’
‘c’ ^ 210
mask = a << 31
```

Precedencia general de operadores

1. ()
2.++, --, + (unitaria), – (unitaria), !
3.*, /, %
4.+, - (suma, resta)
5.>>,<<
6.<, >, <=, >=
7.==, !=
8.&, ^, |,
9.&&,
10.||
11.= (asignación y sus variantes: += -= *= /= <<= >>= %= &= ^= |=)

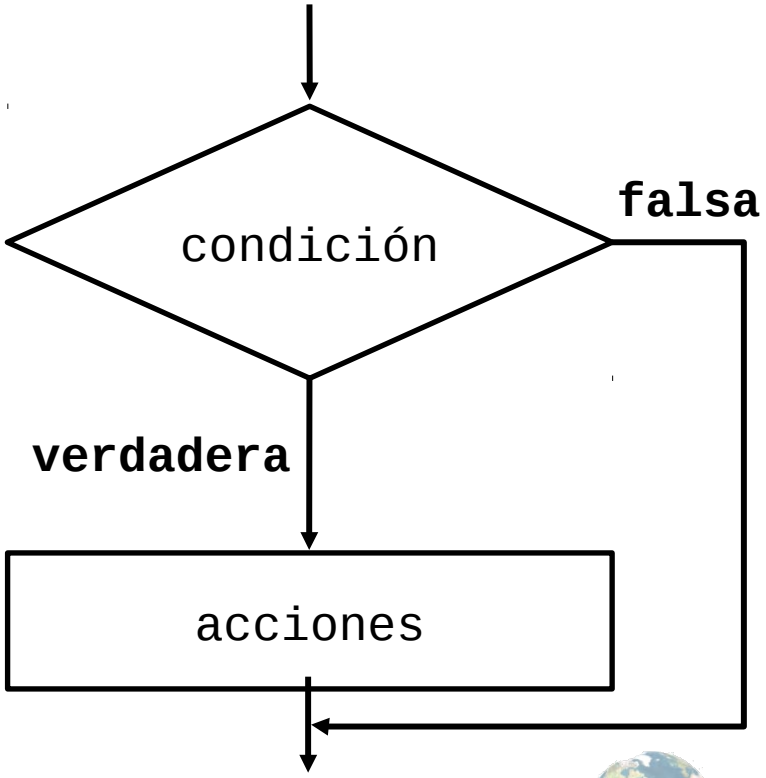
2E – Control del flujo de ejecución

Sentencias, bloques y estructuras

- Sentencias simples: son instrucciones individuales, que no implican alteración del flujo de ejecución (es decir, que no alteran el orden en que se ejecutan las instrucciones)
- Bloque de sentencias: es un conjunto de sentencias que se ejecutan como bloque, generalmente como parte de una estructura de control de flujo; los bloques de sentencias se delimitan con **{** y **}**
- Estructuras de control del flujo de programa
  - Estructuras condicionales: permiten ejecutar un bloque de sentencias sólo cuando se da cierta condición
  - Estructuras iterativas: permiten ejecutar un bloque de sentencias repetidas veces, hasta que se cumple una condición de salida
  - Saltos: permiten saltar directamente a una instrucción determinada **(no recomendado)**
  - Subrutinas: permiten conmutar la ejecución a un subprograma (se verá más adelante)

Condicional simple

```
if([expresión de condición]){
   [bloque de sentencias]
}
```



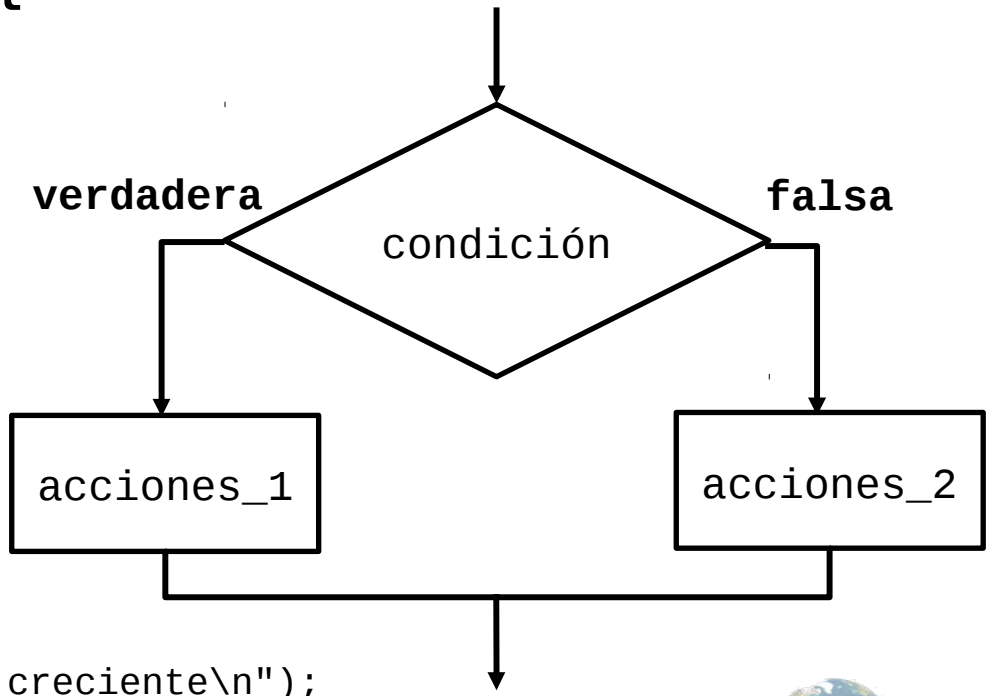

```
#include <stdio.h>

int main(){

    int num1, num2;
    printf("Ingrese 2 numeros en orden creciente\n");
    scanf("%d%d", &num1, &num2 );
    if (num1 < num2 ) {
        printf("Perfecto!");
    }
    return 0;
}
```

Condicional doble

```
if([expresión de condición]){
   [bloque si verdadero]
} else {
   [bloque si falso]
}
```




```
#include <stdio.h>

int main(){

     int num1, num2;
     printf("Ingrese 2 numeros en orden creciente\n");
     scanf("%d%d", &num1, &num2 );
     if (num1 < num2 ) {
          printf("Perfecto!");
     } else {
          printf("Incorrecto! (%d >= %d)",num1,num2);
     }
     return 0;
}
```

Condicional múltiple (opción 1)

```
if([expresión de condición 1]){
  [bloque si verdadero exp1]
} else if([expresión de condición 2]) {
  [bloque si verdadero exp2]
} else {
  [bloque si falso]
}
```

```
#include <stdio.h>

int main(){

    int num1, num2;
    printf("Ingrese 2 números\n");
    scanf("%d%d", &num1, &num2 );
    if (num1 < num2 ) {
        printf("El primero es menor");
    } else if (num1 < num2) {
        printf("El segundo es menor");
    } else {
        printf("Los números son iguales");
    }
    return 0;
}
```

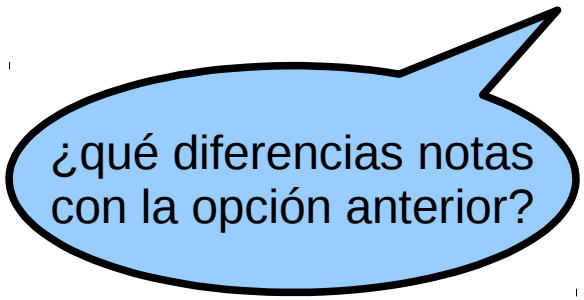
Condicional múltiple (opción 2)

```
switch(expresion) {

   case expresion_constante1  :
      statement(s);
      break; /* interrumpe (opcional) */

    case expresion_constanteN  :
      statement(s);
      break; /* interrumpe (opcional) */

   default : /* opcion por defecto */
      statement(s);
}
```




```
#include <stdio.h>

int main(){

     char grupo;
     printf("Ingrese grupo sanguíneo\n");
     scanf("%c", &grupo );
     switch(grupo) {
       case 'a' :
       case 'A' :
           printf("Grupo A" );
           break;
       case 'b' :
       case 'B' :
           printf("Grupo B" );
           break;
       case '0' :
           printf("Grupo 0" );
           break;
       default :
           printf("Grupo desconocido\n" );
       }
     return 0;
}
```

Condicionales anidados

```
#include <stdio.h>

int main(){

     int nota;
     printf("Ingrese nota (1 al 10)\n");
     scanf("%d", &nota);
     if(nota >= 6) {
          printf("Se encuentra aprobado");
     } else {
          char ptp;
          printf("Presento trabajos practicos? (s/n)\n");
          scanf(" %c",&ptp);
          switch(ptp) {
               case 's':
                    printf("Accede al recuperatorio");
                    break;
               case 'n':
                    printf("No accede al recuperatorio");
                    break;
               default:
                    printf("Respuesta incorrecta");
          }
     }
     return 0;
}
```

Bucles

- Control del flujo de ejecución
  - Estructuras de control iterativas. 3 tipos:
    - Bucle *for*

```
for(inicializacion; condicion de continuidad; actualizacion){
    [bloque de sentencias]
}
```

Ejemplo:

```
int i;
for(i=0; i<10; i++){
    printf("Valor de i: %d", i);
}
```

Bucles

- Control del flujo de ejecución
  - Estructuras de control iterativas. 3 tipos:
    - Bucle *do/while*

```
do{
    [bloque de sentencias]
}while(condicion de continuidad);
```

    - Bucle *while*

```
while(condicion de continuidad){
    [bloque de sentencias]
}
```

Unidad 2E – Control del flujo de ejecución

- Control del flujo de ejecución
  - Estructuras de control anidadas
    - Cualquiera de las estructuras vistas se puede anidar dentro de la otra estructura, sin limites:
      - Ej:
        - Podemos escribir un if/else dentro de un bucle for
        - Podemos escribir un bucle while dentro de un if/else
        - Podemos escribir un bucle while dentro de un bucle for que a su vez está dentro de una sentencia condicional
        - Podemos escribir un for dentro de otro for
        - Etc.
  - Existen palabras reservadas especiales que permiten alterar el flujo de programa de manera más específica: `break`, `continue` y `goto`

Unidad 2E – Control del flujo de ejecución

- Control del flujo de ejecución
  - `break:` corta la ejecución de un **bucle** sin ejecutar el resto de la vuelta actual, tal y como si no se hubiera cumplido la condición de continuidad
  - `continue:` pasa a la siguiente vuelta de un **bucle**, sin ejecutar el resto de la vuelta actual
  - `goto:` permite saltar a una instrucción específica, denotada por una etiqueta.
    - Esto no se recomienda, por ser muy propenso a errores (va en contra de los principios de la programación estructurada)
    - Tiende a genera “código spaghetti”

2F – Estructuras de datos

Estructuras de datos

- Un arreglo es una estructura de datos indizada
- Los elementos de un arreglo se almacenan en posiciones de memoria contiguas
- Todos los elementos de un arreglo son del mismo tipo de dato
- Sintaxis:
  - Declaración

```
int arreglo[10]; //Vector de 10 elementos
```

  - Asignación

```
arreglo[2] = 5; //Guardamos 5 en el elemento 2
```

Estructuras de datos

- Los arreglos pueden tener cualquier cantidad de dimensiones. Ej
  - 2D

```
int arreglo[10][10];
arreglo[2][2] = 5;
```

  - 3D

```
int arreglo[10][20][30];
arreglo[1][2][3] = 5;
```

  - 4D

```
int arreglo[10][12][3][2];
arreglo[1][2][3][0] = 5;
```

Estructuras de datos

- Las cadenas de caracteres en C se representan con un arreglo de char, terminado con el carácter especial \0.
- El carácter de fin de cadena ocupa 1 espacio, por lo que un arreglo de N elementos guarda cadenas de hasta N-1 caracteres
- Ej:

```
char cadena[10]; //Cadena de 9 elementos
cadena[0] = 'a';
cadena[1] = 'b';
…
cadena[8] = 'h';
cadena[9] = '\0';
printf(“%s”, cadena);
```

Estructuras de datos

- Tipos definidos por el usuario
  - Además de los tipos básicos (int, char, etc.), pueden definirse tipos personalizados. Hay 2 casos:
    - Definición de tipos personalizados simples
    - Definición de tipos complejos
  - Tipos de datos simples: se definen como un “alias” de otro tipo simple ya existente.
    - Ej: definimos un tipo “boolean” para guardar valores lógicos true/false, algo que no existe en C

      `typedef int boolean;` → definimos el tipo “boolean”

      ```
      #define TRUE 1
      #define FALSE 0
      ```

Estructuras de datos

- Tipos definidos por el usuario
  - Un tipo de datos complejo es una estructura que agrupa varias variables, almacenándolas en posiciones de memoria contiguas
  - Es similar a un arreglo, sólo que permite almacenar variables de distintos tipos
  - Definición de una estructura:

```
struct VERTICE{
   double x;
   double y;
};
```

Estructuras de datos

- Tipos definidos por el usuario
  - Creación de una *instancia* de la estructura:

```
struct VERTICE v1;
```

  - Acceso a los elementos de la estructura: operador .

```
v1.x = 10;  //Asigna el valor (10, 10)
v1.y = 10;   //al vertice
printf("(%f, %f)", v1.x, v1.y);
```

  - `typedef` también puede usarse para estructuras

```
typedef struct VERTICE vertice_grafico;
vertice_grafico v1;
```