UDLAP
UNIVERSIDAD DE LAS AMÉRICAS PUEBLA

Lenguaje C: Repaso

Dr. José Luis Zechinelli Martini
joseluis.zechinelli@udlap.mx

LDS – 1101

Gracias a Ángel Salas, Universidad de Zaragoza

Plan

- Estructura y formato de un programa
- Elementos léxicos
- Expresiones y asignación
- Tipos de datos fundamentales
- Instrucciones de control
- Tipos de datos estructurados

Programa (1)

- Un programa escrito en C está formado por una o varias funciones
- Cada función expresa la realización del algoritmo que resuelve una de las partes en que se ha descompuesto un problema:
  - En el método de análisis descendente, un problema complejo se descompone sucesivamente en partes cada vez más simples hasta que resulta trivial su programación
  - Con el lenguaje C, cada una de esas partes se resuelve con una función

Programa (2)

➔ Las funciones que componen un programa pueden definirse en el mismo archivo o en archivos diferentes

- Una función tiene un nombre y está formada por un conjunto de instrucciones que se ejecutan devolviendo un valor a quien la invoca

- Todo programa debe contener la función "main", la cual es invocada desde el sistema operativo cuando comienza la ejecución del programa:
  - Controla la ejecución del resto del programa
  - Si no es de tipo "void", devuelve un valor al sistema cuando termina la ejecución
  - La terminación de la función "main" se puede controlar con la función de biblioteca "exit"

El menor de dos enteros leídos

```
#include <stdio.h>

int menor( int a, int b )
{
      if( a < b ) return a;                    // Los cuerpos del then y del else están formados
      else return b;                           // por una sola instrucción
}

void main()
{
      int n1, n2;

      printf( "Introducir dos enteros:\n" );
      scanf( "%d%d", &n1, &n2 );                          /* Aquí se leen dos valores enteros */

      if( n1 == n2 )
            printf( "Son iguales\n" );
      else
            printf( "El menor es: %d\n", menor(n1, n2) );          /* Imprime un entero */
}
```

Definición de una función

- Formato general:

```
tipo nombre ( lista_de_parámetros )
{
        cuerpo
}
```

- Visión detallada con tres parámetros y tres variables:

```
tipo nombre( tipo p1, tipo p2, tipo p3 )
{
        tipo v1, v2, v3;
        Instrucción1;
        ...
        return (expresión);
}
```

➔ Todas las instrucción termina con ";" incluso la declaración de variables

Definiciones de variables

- Todas las variables deben declararse antes de usarse
- Esto determina:
  - Su representación en memoria
  - El conjunto de valores posibles
  - Las operaciones permitidas
  - La forma de realizar las operaciones
- También se puede indicar:
  - La clase de almacenamiento en memoria (ver tipos de datos fundamentales)
  - El ámbito de existencia (las locales sólo existen dentro de la función que las define)
  - Un valor inicial

Reglas de alcance (1)

- Las variables declaradas fuera del cuerpo de una función son variables globales
- Se pueden poner declaraciones al principio de cualquier bloque:

```
{
        int  i, j, *p;
        float  r, f;
        char  c, s[40];
        …
}
```

- Las variables sólo son conocidas dentro del bloque en que han sido declaradas y en sus sub-bloques (bloques de un bloque)

Reglas de alcance (2)

```
int g;                          /* "g" es una variable global */
void main( )
{
    int  a, b;
    {
        float  b, x, y;         // Son accesibles "a", pero no "b"
        …                       // "b" ha quedado enmascarada
    }
    {
        unsigned  a;
        char  c, d;             // Es alcanzable "b", pero no "a"
        …                       // No son accesibles "x" e "y"
    }
                                // Son alcanzables "int a" e "int b"
    …                           // No son alcanzables "x", "y", "c"
}
```

➔ La variable "g“ es alcanzable en todos los bloques

Función: printf (1)

- Permite escribir con formato:
  - A través del dispositivo de salida estándar
  - La función "printf" convierte, formatea e imprime

- Admite dos tipos de parámetros:
  - La especificación del formato
  - La lista de valores o variables
    - Se dan los valores que hay que imprimir y el formato de conversión

Función: printf (2)

- Con el formato de conversión se indica el tipo de transformación:
  - %d formatea el valor para imprimir un decimal
  - %o formatea el valor para imprimir un octal sin signo
  - %x formatea el valor para imprimir un hexadecimal sin signo
  - %f considera que el valor es de tipo "float" o "double" para imprimir un número real
  - %e considera que el valor es de tipo "float" o "double" para imprimir un número real usando notación exponencial
  - %c considera que el valor es un carácter simple
  - %s considera que el valor es una cadena de caracteres y la formatea como una secuencia de caracteres

Función: printf (3)

- Los valores que pueden ponerse para imprimir pueden ser representados por constantes, variables, llamadas a funciones o cualquier otra expresión
- Ejemplos:

```
printf( "El número %d es impar.\n", n );

for( r = 3.1416; r >= 0.0; r = r – 0.001 )

        printf ( "%7.4f\n", r );

printf ( "Atención!! %c\n", 7 );

printf ( "Potencia: %f\n", potencia(base, exponente) );
```

Plan

- ✓ Estructura y formato de un programa
- Elementos léxicos
- Expresiones y asignación
- Tipos de datos fundamentales
- Instrucciones de control
- Tipos de datos estructurados

Palabras reservadas

- Son algunos símbolos cuyo significado está predefinido y no se pueden usar para otro fin:

auto	break	case	char
continue	default	do	double
else	enum	extern	float
for	goto	if	int
long	register	return	short
sizeof	static	struct	switch
typedef	union	unsigned	void
while			

Constantes

- Son entidades cuyo valor no se modifica durante la ejecución de un programa
- Hay varios tipos de constantes :
  - **Numéricas**: -7, 3.1416, 2.5e-3
  - **Caracteres**: 'a', '\n', '\0'
  - **Cadenas**: "índice general"
- Constantes simbólicas: #define PI 3.1416
  - Se definen con instrucciones para el preprocesador
  - Se define un identificador y se le asigna un valor constante
  - Favorecen la realización de modificaciones en los programas

Cadenas

➔ Para realizar operaciones con cadenas hay que usar funciones, el lenguaje no dispone de operadores para manipular cadenas

- Es una secuencia de caracteres, almacenados en posiciones de memoria contiguas (arreglo), que termina con el carácter nulo
- Una cadena se representa escribiendo una secuencia de caracteres encerrada entre comillas:
  - "buenos días"
  - "así: \" se pueden incluir las comillas"
- Una cadena es un valor constante de una estructura de tipo "array of char":
  - char titulo[24];
  - strcpy(titulo, "Curso de C");
  - printf("%s", titulo);

Precedencia de operadores (1)

Operadores	Asociatividad
( ) [ ] -> . (miembro)	izquierda a derecha
~ ! ++ -- sizeof (tipo) -(unario) *(indirección) &(dirección)	derecha a izquierda
* / %	izquierda a derecha
+ -	izquierda a derecha
<< >>	izquierda a derecha
< <= > >=	izquierda a derecha
== !=	izquierda a derecha
…	…

Precedencia de operadores (2)

Operadores	Asociatividad
…	…
&	izquierda a derecha
^	izquierda a derecha
|	izquierda a derecha
&&	izquierda a derecha
||	izquierda a derecha
?:	derecha a izquierda
= += -= *= ...	derecha a izquierda
, (operador coma)	izquierda a derecha

Plan

- ✓ Estructura y formato de un programa
- ✓ Elementos léxicos
- Expresiones y asignación
- Tipos de datos fundamentales
- Instrucciones de control
- Tipos de datos estructurados

Expresiones de asignación

- Se forman con el operador "=": variable = expresión
  - Se evalúa la expresión que está a la derecha y
  - El resultado se asigna a la variable del lado izquierdo
- Ejemplos:
  - n = 5 // la expresión toma el valor 5
  - x = (y=5) + (z=7) // y toma el valor 5
    // z toma el valor 7
    // x toma el valor 12
- El operador de asignación "=" es asociativo de derecha a izquierda:

  x = y = z = 0 es equivalente a x = (y = (z = 0))

Plan

- ✓ Estructura y formato de un programa
- ✓ Elementos léxicos
- ✓ Expresiones y asignación
- Tipos de datos fundamentales
- Instrucciones de control
- Tipos de datos estructurados

Tipos de datos (2)

- Atómicos:
  - Entero: short, int, long, unsigned short, unsigned, unsigned long
  - Real: float, double
  - Carácter: char
  - Apuntador: * (pointer)
  - Enumerativos: enum
- Complejos:
  - arreglo
  - struct
  - union
  - campos de bits

Clases de almacenamiento

- auto (*automatic*):
  - Son variables locales al entrar a un bloque
  - Desaparecen al terminar la ejecución del bloque
- static:
  - Locales de un bloque
  - Conservan su valor entre ejecuciones sucesivas del bloque
- extern:
  - Existen y mantienen sus valores durante toda la ejecución del programa
  - Pueden usarse para la comunicación entre funciones, incluso si han sido compiladas por separado
- register:
  - Se almacenan, si es posible, en registros de alta velocidad de acceso
  - Son locales al bloque y desaparecen al terminar su ejecución

Inicialización de variables

- Por defecto las variables:
  - extern y static se inicializan a cero
  - auto y register quedan indefinidas (valores aleatorios)
- Se pueden asignar valores iniciales en la declaración de las variables atómicas: tipo variable = valor;
  - Con las variables extern y static:
    - Se hace una vez, en tiempo de compilación
    - Se deben usar valores constantes
  - Con las variables auto y register:
    - Se hace cada vez que se entra en el bloque
    - Se pueden usar variables y llamadas a funciones

Número de bytes: Sizeof

```
# include <stdio.h>
void main()
{
      char c;
      short s;
      int i;
      long l;
      float f;
      double d;
      printf( "Tipo char: %d bytes\n", sizeof(c) );
      printf( "Tipo short: %d bytes\n", sizeof(s) );
      printf( "Tipo int: %d bytes\n", sizeof(i) );
      printf( "Tipo long: %d bytes\n", sizeof(l) );
      printf( "Tipo float: %d bytes\n", sizeof(f) );
      printf( "Tipo double: %d bytes\n", sizeof(d) );
}
```

Tipos atómicos

```
# include <stdio.h>

void main()
{
      printf( "\n char: %d bytes", sizeof(char) );
      printf( "\n short: %d bytes", sizeof(short) );
      printf( "\n unsigned short: %d bytes", sizeof(unsigned short) );
      printf( "\n int: %d bytes", sizeof(int) );
      printf( "\n unsigned: %d bytes", sizeof(unsigned) );
      printf( "\n long: %d bytes", sizeof(long) );
      printf( "\n unsigned long: %d bytes", sizeof(unsigned long) );
      printf( "\n float: %d bytes", sizeof(float) );
      printf( "\n double: %d bytes", sizeof(double) );
      printf( "\n\n" );
}
```

Enteros en arquitecturas a 32 bits

	Bits	Mínimo	Máximo
int	32	-2^{31} (-2,147,483,648)	$2^{31} - 1$ (+2,147,483,647)
unsigned	32	0	$2^{32} - 1$ (+4.294.967.295)
short	16	-2^{15} (-32,768)	$2^{31} - 1$ (+32,767)
unsigned short	16	0	$2^{16} - 1$ (+65,535)
long	32	-2^{31} (-2,147,483,648)	$2^{31} - 1$ (+2,147,483,647)
unsigned long	32	0	$2^{32} - 1$ (+4.294.967.295)
char	8	0	255

Plan

- ✓ Estructura y formato de un programa
- ✓ Elementos léxicos
- ✓ Expresiones y asignación
- ✓ Tipos de datos fundamentales
- Instrucciones de control
- Tipos de datos estructurados

Instrucción vacía y compuesta

- Instrucción vacía:
  - Sólo la instrucción ";"
  - Ciclo infinito: for(; ;);

- Instrucción compuesta:
  - Una secuencia de instrucciones encerradas entre llaves

```
{
        int i;
        float p = 1.0;
        for( i=1; i <= e; ++i )
                p = p * b;
        return p;
}
```

  - Una instrucción compuesta se puede poner en cualquier lugar donde pueda ir una simple

Instrucción " if – else " (2)

- Cuando el compilador lee varias instrucciones "if" anidadas: asocia cada "else" con el "if" más próximo:

```
if( c == ' ' )
  ++blancos;
else if( '0' <= c && c <= '9' )
  ++cifras;
else if( 'a' <= c && c <= 'z' || 'A' <= c && c <= 'Z' )
  ++letras;
else if( c == '\n' )
  ++saltos;
else
  ++otros;
```

El operador condicional " ? "

- Es un operador ternario: exp1 ? exp2 : exp3
  - Se evalúa la expresión "exp1"
  - Si tiene valor distinto de cero, se evalúa la expresión "exp2" y su resultado será el valor de la expresión
  - Si "exp1" tiene valor cero, se evalúa "exp3" y su resultado será el valor de la expresión

- La instrucción:
  - x = (y < z) ? y : z ;
  - Tiene un efecto equivalente a:

```
if( y < z )
        x = y;
else
        x = z;
```

Ejemplo del operado " ? "

- Imprimir los elementos de un "array" de "int" poniendo cinco elementos en cada línea:

```
#define TAM 10

void main()
{
        int a[ TAM ] = { 1, 2, 3, 4, 5, 6, 7, 8, 9, 10 };

        for( i = 0; i < TAM; ++i )
                printf( "%c%d", ( i%5 == 0 ) ? '\n' : '\0', a[i] );
}
```

La instrucción " while "

- Inicializar un arreglo:

```
float v[100];
int n;
n=0;
while( n < 100 ) {
        v[n] = 0.0f;
        ++n;
}
```

- Leer un carácter mientras no sea “fin de archivo”:

```
while( scanf( "%c\n", &c ) != EOF )
        printf( "El carácter leído es: %c\n", c );
```

La instrucción " for "

- for(expresion1; expresion2; expresion3) instrucción;
- Su semántica es equivalente a la de:

```
expresion1;
while( expresion2 ) {
        instrucción;
        expresion3;
}
```

- Pueden faltar alguna o todas las expresiones:
  - Deben permanecer los signos de punto y coma:

```
i = 1; suma = 0;
for( ; i <= 10; ++i ) suma += i;
```

  - Es equivalente a:

```
i = 1; suma = 0;
for( ; i <= 10; ) suma += i++;
```

La instrucción " do "

- do instrucción while(expresión);
  - Es una variante de "while"
  - El test se hace al final del ciclo

```
/* Se repite la orden de lectura mientras el valor introducido sea
   menor o igual que cero */
{
        int n;
        do {
                printf( "Entero positivo ? " );
                scanf( "%d", &n );
        } while( n <= 0 );
}
```

La instrucción " switch " (1)

- Es una instrucción condicional múltiple que generaliza a la instrucción "if – else":

```
switch( expresion_entera ) {
        case expresion_constante_entera:  instrucción(es)
                                          break;
        case expresion_constante_entera:  instrucción(es)
                                          break;
        . . .
        default:                          instrucción(es)
}
```

  - Se evalúa la "expresion_entera" y se ejecuta la clausula "case" que corresponda con el valor obtenido
  - Si no se encuentra correspondencia, se ejecuta el caso "default"; y si no lo hay, termina la ejecución de "switch"
  - Termina cuando se encuentra una instrucción "break"

La instrucción " switch " (2)

```
c = getchar( );
while( c != '.' && c != EOF ) {
    switch( c ) {
        case 'c' :      consultas();
                        break;
        case 'a' :      altas();
                        break;
        case 'b' :      bajas();
                        break;
        case 'm' :      modificaciones();
                        break;
        default :       error();
    }
    c = getchar();
}
```

La instrucción " break "

- Fuerza la salida inmediata de un ciclo, saltando la evaluación de la condición normal:

```
# include <math.h>
...
while( 1 ) {
        scanf( "%f", &x );
        if( x < 0.0 ) break;
        printf( "%f\n", sqrt(x) );
}
```

- Termina una clausula "case" de la instrucción "switch" y finaliza su ejecución:

```
switch( c ) {
        case '.' :   mensaje( "fin" );
                     break;
        case '?' :   ayuda();
                     break;
        default :    procesar( c );
}
```

La instrucción " continue "

- Se detiene la iteración actual de un ciclo y se inicia la siguiente evaluando la condición:

```
for( i = 0; i < TOTAL; ++i ) {
        c = getchar( );
        if( '0' <= c && c <= '9' )
                continue;
        procesar( c );
}
```

Plan

- ✓ Estructura y formato de un programa
- ✓ Elementos léxicos
- ✓ Expresiones y asignación
- ✓ Tipos de datos fundamentales
- ✓ Instrucciones de control
- ■ Tipos de datos estructurados

Tipos de datos estructurados

- Pueden contener más de un componente simple o estructurado:
  - Arreglos (operador "[ ]")
  - Estructuras (operador "struct")
  - Campos de bits (operador "struct")
  - Compartir memoria (operador "union")
- Se caracterizan por:
  - El tipo o los tipos de los componentes
  - La forma de la organización
  - La forma de acceso a los componentes
- Enseguida estudiaremos a los arreglos y a las estructuras

Arreglos (1)

- Una organización de datos caracterizada por:
  - Todos los componentes son del mismo tipo (homogéneo)
  - Acceso directo a sus componentes
  - Todos sus componentes se pueden seleccionar arbitrariamente y son igualmente accesibles

```
int v[100], a[100][100];

for( i = 0; i < 100; i++ )
        v[ i ] = 0;                          /* Arreglo unidimensional */

for( i = 0; i < 100; i++ )
        for( j = 0; j < 100; j++ )
                a[ i ][ j ] = 0;             /* Arreglo bidimensional */
```

Arreglos (2)

- Declaración: tipo identificador [tamaño];
  - El "tamaño" tiene que ser una expresión entera positiva que indica el número de elementos del arreglo
  - Los elementos se identifican escribiendo el nombre de la variable y un índice escrito entre corchetes:
    - v[0] v[1] v[2] ... v[8] v[9]
    - Límite inferior del índice = 0
    - Límite superior del índice = tamaño – 1
- El uso de constantes simbólicas facilita la modificación de los programas:

```
#define TAM 10
int a[ TAM ];
```

Inicialización de arreglos

- Asignación de valor en la sentencia de declaración:
  - Se pueden inicializar los de clase "static" o "extern"
  - No se pueden inicializar los de clase "auto"
- Sintaxis: clase tipo identificador[tamaño] = { lista de valores };

  static float v[5] = { 0.1, 0.2, -1.7, 0.0, 3.14 };
- Si se escriben menos valores, el resto se inicializan a cero:

  static int a[20] = { 1, 2, 3, 4, 5, 6 };

  static char s[24] = "los tipos de datos";
- Si no se declaró tamaño, se deduce de la inicialización:

  static int a[ ] = { 0, 0, 0 };

El índice

- Los elementos de un arreglo responden a un nombre de variable común y se identifican por el valor de una expresión entera, escrita entre corchetes:
  - El índice debe tomar valores positivos e inferiores al indicado en la declaración
  - Los desbordamientos no son monitoreados durante la ejecución
  - Si el índice sobrepasa el límite superior, se obtiene un valor incorrecto porque se está haciendo referencia a una posición de memoria ajena
  - Modificando el valor del índice se puede recorrer toda la estructura

```
# define TAM 100
int i, suma, a[ TAM ];
suma = 0;
for( i = 0; i < TAM; ++i ) suma += a[ i ];
```

Inicialización y manipulación (1)

```
# include <stdio.h>

void main()
{
    auto int i, j;
    static int enteros[5] = { 3, 7, 1, 5, 2 };
    static char cadena1[16] = "cadena";
    static char cadena2[16] = { 'c', 'a', 'd', 'e', 'n', 'a', '\0' };
    static char *c = "cadena";
    static int a[2][5] = { { 1, 22, 333, 4444, 55555 } , { 5, 4, 3, 2, 1 } };
    static int b[2][5] = { 1, 22, 333, 4444, 55555, 5, 4, 3, 2, 1 };
    static struct {
            int i;
            float x;
    } sta = { 1, 3.1415e4 }, stb = { 2, 1.5e4 };
```

Inicialización y manipulación (2)

```
static struct {
        char c;
        int i;
        float s;
} st[2][3] = { {{ 'a', 1, 3e3 }, { 'b', 2, 4e2 }, { 'c', 3, 5e3 }}, {{ 'd', 4, 6e2 },} };

printf( "enteros:\n" );
for( i = 0; i < 5; ++i )
        printf( "%d ", enteros[i] );

printf( "\n\n" );
printf( "cadena1:\n%s\n\n", cadena1 );
printf( "cadena2:\n%s\n\n", cadena2 );
printf( "\n\n" );
printf( "c:\n%s\n\n", c );
```

Inicialización y manipulación (3)

```
    printf( "a:\n" );
    for( i = 0; i < 2; ++i )
            for( j = 0; j < 5; ++j )
                    printf( "%d ", a[i][j] );
    printf( "\n\n" );
    printf( "b:\n" );
    for( i = 0; i < 2; ++i )
            for( j = 0; j < 5; ++j )
                    printf( "%d ", b[i][j] );
    printf( "sta:\n" );
    printf( "%d %f \n\n", sta.i, sta.x );
    printf( "st:\n" );
    for( i = 0; i < 2; ++i )
            for( j = 0; j < 3; ++j )
                    printf( "%c %d %f\n", st[i][j].c, st[i][j].i, st[i][j].s );
}
```

Arreglos y funciones

- Cuando se pasa como parámetro un arreglo:
  - Sólo se pasa su dirección
  - Los elementos del arreglo no se copian

- Definición de parámetros:

```
int sumar( int v[ ] , n )
{
        int i, s = 0;
        for( i = 0; i < n; ++i )
                s += v[i];
        return s;
}
```

```
int sumar( int * v, n )
{
        . . .
        . . .
        . . .
        . . .
}
```

Estructuras (1)

- Las estructuras son organizaciones de datos cuyos miembros (campos) pueden ser de tipos diferentes

- Ejemplo:

```
struct naipe {
    int valor;
    char palo;
};

struct naipe carta, c;
```

Estructuras (2)

```
…
carta.valor = 10;
carta.palo = 'e';
c = carta;
enum palos { oros, copas, espadas, bastos };

struct naipe {
      int valor;
      enum palos palo;
};

carta.palo = espadas;
```

Miembro de estructura " . "

- El descriptor de un miembro de estructura tiene la forma:

  variable_estructura . nombre_miembro

  - Un nombre de miembro no se puede repetir en una estructura
  - Puede haber estructuras diferentes que tengan miembros del mismo nombre

```
struct complejo {
        float real;
        float imaginaria;
} x, y, z, a[10][10];

x.real = 2.7;
x.imaginaria = -0.5;
y.real = x.real;
z = x;
a[2][4].real = x.real;
```

Puntero a estructura " -> "

- Es frecuente usar punteros a estructuras:
  - Por ejemplo, en las implementaciones donde no se permite que las funciones devuelvan valor de tipo "struct"
  - Por eso existe un símbolo especial para los punteros a estructuras: -> ("menos" seguido de "mayor que")

- La descripción de un miembro tiene la forma:

  variable_puntero_a_estructura -> variable_miembro

- Recordar que los operadores apuntador a estructura " -> ", miembro de estructura " . ", paréntesis " ( ) " y corchetes " [ ] " tienen:
  - La prioridad más alta
  - Asociatividad de izquierda a derecha

Ejemplo

```
struct alumno {
    int curso;
    char grupo;
    char nombre[40];
    float notas[10];
};
```

Expresión "."	Expresión "->"	Valor
buf.curso	p->curso	2
buf.grupo	p->grupo	'A'
buf.nombre	p->nombre	"Pedro Pérez"
buf.notas[5]	p->notas[5]	7.5

```
struct alumno buf, * p = & buf;
buf.curso = 2;
buf.grupo = 'A';
strcpy( buf.nombre, "Pedro Perez");
buf.notas[5] = 7.5;
```

Estructuras y funciones

- Las estructuras se pueden pasar por valor a las funciones
- Las funciones pueden devolver un valor de tipo "struct" (depende de la implementación)
- Aunque la implementación no permita declarar funciones de tipo "struct", siempre se pueden declarar como apuntadores

```
struct complejo {
        float re;
        float im;
};
```