Presentación de EDyAI

Con una introducción a C

E. Rivas

Marzo de 2015

Presentación

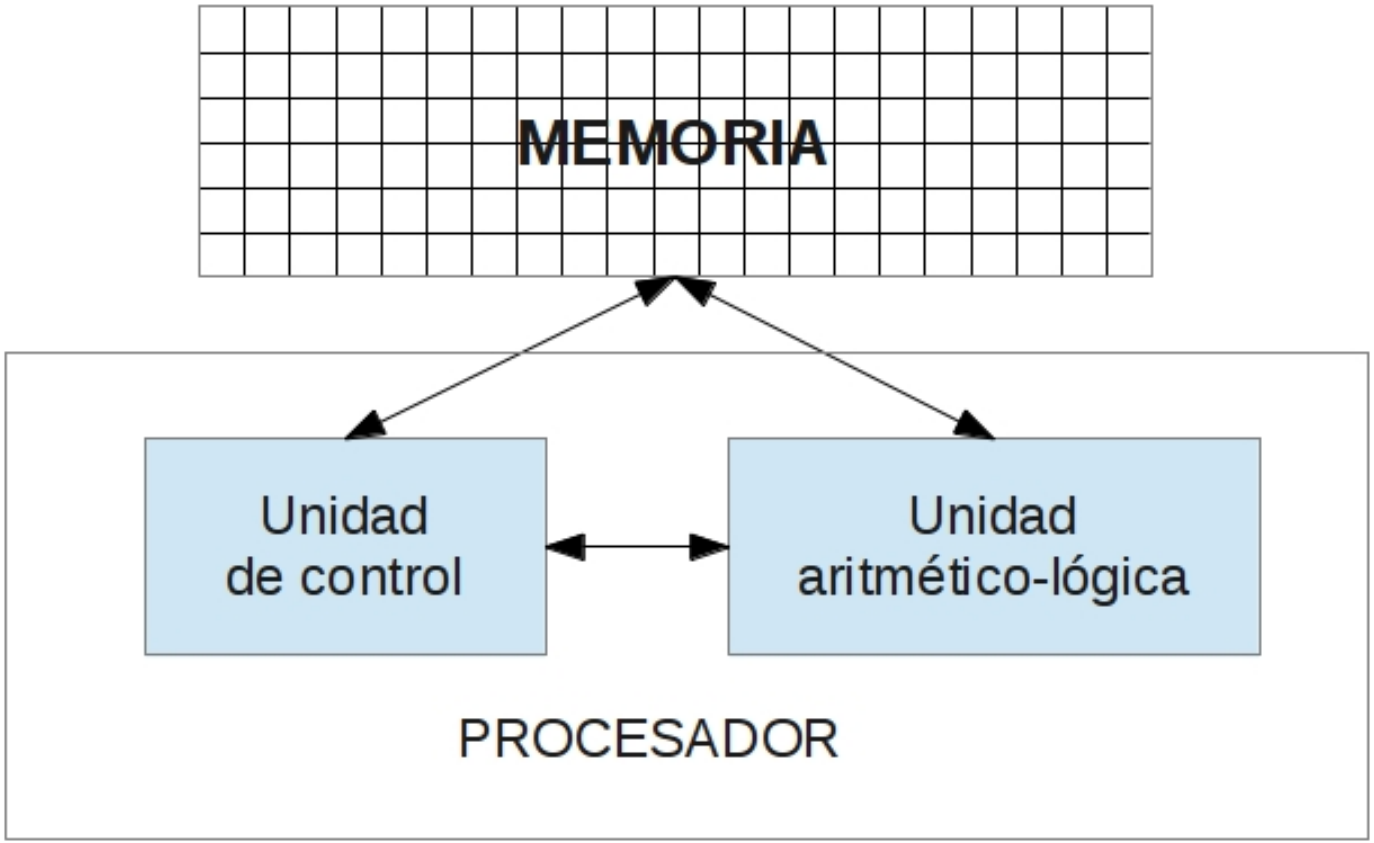
- Nombre: Estructuras de Datos y Algoritmos I.
- Temas:
  - Estructuras de datos: listas, árboles, etc.
  - Algoritmos: ordenamiento, búsqueda, etc.
  - Programación en C.
- Horarios:
  - Lunes: 10:30 a 12:45 (lab 2)
  - Martes: 7:30 a 9:30 (lab 1)
  - Viernes: 10:30 a 12:45 (aula 05)

¿Qué son las estructuras de datos? ¿Y los algoritmos?

Algoritmo Un conjunto preescrito de instrucciones precisas, ordenadas y finitas que permite realizar una tarea mediante pasos sucesivos.

Estructura de datos Mecanismo de organización de datos que provee una manera conveniente y eficiente para acceder y manipularlos.

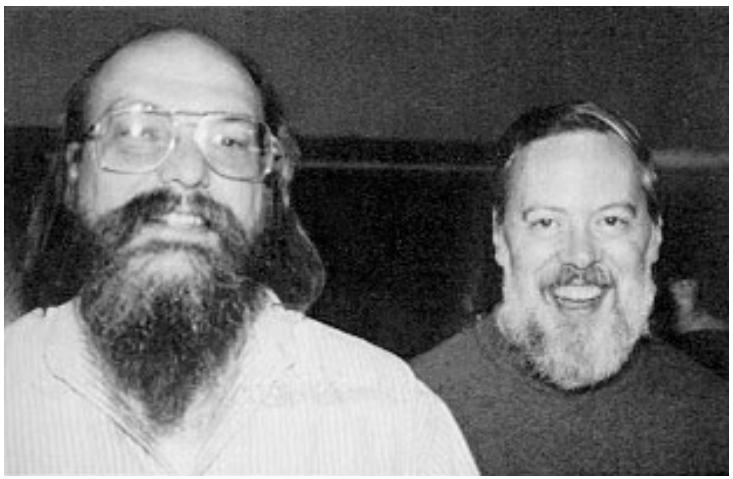
Modelo computacional



Lenguaje de programación C

Programaremos en el lenguaje de programación C.

- Surge al rededor de 1970.
- Es desarrollado en los Bell Labs.
- Es un lenguaje imperativo.
- Creado por Dennis Ritchie.



Un “hola mundo”

```
#include <stdio.h>

int main(void) {
  printf("Hola mundo.\n");
  return 0;
}
```

Un ejemplo más avanzado

```
#include <stdio.h>

int main(void) {
  int x;
  x = 42;
  x = x * 2;
  printf("El doble de 42 es %d.\n", x);
  return 0;
}
```

Tipos de datos básicos

Nombre	Valor min.	Valor max.	Ej. literal
`char`	0	255	`'A'`
`int`	-32767	+32767	42
`long`	-2147483647	+2147483647	147483647L
`float`			1.5f
`double`			1.5

Declaración de variables

Las variables declaran al comienzo de las funciones (en particular, de `main`), con el formato:

```
tipo variable;
```

Declaración de variables

Las variables declaran al comienzo de las funciones (en particular, de `main`), con el formato:

```
tipo variable;
```

Por ejemplo, las siguientes declaraciones:

```
int x;
float p;
char c;
```

Proposición

Luego de declarar las variables en un cuerpo de función, se procede a escribir una lista de proposiciones. Estas especifican las operaciones de cálculo que se van a realizar.
Algunas proposiciones son:

Proposición

Luego de declarar las variables en un cuerpo de función, se procede a escribir una lista de proposiciones. Estas especifican las operaciones de cálculo que se van a realizar.
Algunas proposiciones son:

- asignación.

Proposición

Luego de declarar las variables en un cuerpo de función, se procede a escribir una lista de proposiciones. Estas especifican las operaciones de cálculo que se van a realizar.
Algunas proposiciones son:

- asignación.
- `if`.

Proposición

Luego de declarar las variables en un cuerpo de función, se procede a escribir una lista de proposiciones. Estas especifican las operaciones de cálculo que se van a realizar.
Algunas proposiciones son:

- asignación.
- `if`.
- `while`.

Proposición

Luego de declarar las variables en un cuerpo de función, se procede a escribir una lista de proposiciones. Estas especifican las operaciones de cálculo que se van a realizar.
Algunas proposiciones son:

- asignación.
- `if`.
- `while`.
- `for`.

Proposición

Luego de declarar las variables en un cuerpo de función, se procede a escribir una lista de proposiciones. Estas especifican las operaciones de cálculo que se van a realizar.
Algunas proposiciones son:

- asignación.
- `if`.
- `while`.
- `for`.
- `return`.

Proposición

Luego de declarar las variables en un cuerpo de función, se procede a escribir una lista de proposiciones. Estas especifican las operaciones de cálculo que se van a realizar.
Algunas proposiciones son:

- asignación.
- `if`.
- `while`.
- `for`.
- `return`.
- nula.

Asignación

Cuando deseamos sobre-escribir el valor de una variable, utilizamos la sentencia de asignación:

```
var = expresión;
```

Asignación

Cuando deseamos sobre-escribir el valor de una variable, utilizamos la sentencia de asignación:

```
var = expresión;
```

Por ejemplo, las siguientes asignaciones:

```
x = 42;
p = 4.2;
c = 'X';
```

Asignación

Cuando deseamos sobre-escribir el valor de una variable, utilizamos la sentencia de asignación:

```
var = expresión;
```

Por ejemplo, las siguientes asignaciones:

```
x = 42;
p = 4.2;
c = 'X';
```

Podemos inicializar el valor de una variable recién declarada (mientras tanto contiene "basura"):

```
int z = 35;
```

Operadores aritméticos

Los operadores aritméticos se utilizan para formar expresiones de la manera usual.

Entre los operadores disponibles están: `+`, `-`, `*`, `/`, `%`.

Operadores aritméticos

Los operadores aritméticos se utilizan para formar expresiones de la manera usual.

Entre los operadores disponibles están: `+`, `-`, `*`, `/`, `%`.

Por ej.: para calcular el polinomio $4x^2 + 3x - 2$, podemos utilizar la expresión `4*x*x + 3*x - 2`, suponiendo que `x` es una variable previamente definida.

Operadores aritméticos

Los operadores aritméticos se utilizan para formar expresiones de la manera usual.

Entre los operadores disponibles están: `+`, `-`, `*`, `/`, `%`.

Por ej.: para calcular el polinomio $4x^2 + 3x - 2$, podemos utilizar la expresión `4*x*x + 3*x - 2`, suponiendo que `x` es una variable previamente definida.

La expresión evaluará a un valor según el estado del programa. Podremos usar esa expresión en cualquier lugar que vaya un valor. Por ej.:

```
z = x*x + 2;
printf("%d", x + 9 - z);
```

Operadores booleanos

Además de expresiones aritméticas, podemos formar expresiones booleanas.

- C no provee un tipo de datos para booleanos[1].
- Se utilizan expresiones enteras para representar los valores booleanos.
- 0 representará lo falso.
- Cualquier otro valor representará lo verdadero.

[1] Esto no es del todo verdad.

Operadores booleanos

Además de expresiones aritméticas, podemos formar expresiones booleanas.

- C no provee un tipo de datos para booleanos[1].
- Se utilizan expresiones enteras para representar los valores booleanos.
- 0 representará lo falso.
- Cualquier otro valor representará lo verdadero.

Existen operadores correspondientes a las operaciones booleanas clásicas: `&&`, `||`, `!` para representar and, or y not.

[1] Esto no es del todo verdad.

Operadores de comparación

Se pueden comparar valores por alguna relación (igualdad, desigualdad, etc.) y obtener un valor que representa el valor de verdad de la comparación.

Los operadores existentes son: ==, !=, <, <=, >, >=.

Operadores de comparación

Se pueden comparar valores por alguna relación (igualdad, desigualdad, etc.) y obtener un valor que representa el valor de verdad de la comparación.

Los operadores existentes son: ==, !=, <, <=, >, >=.

Por ej. `(x == 4) && (z < 3)` representa una expresión que evalúa a verdadero si en el estado actual `x` toma el valor 4, y `z` toma un valor menor a 3.

Proposiciones compuestas

Podemos agrupar varias proposiciones utilizando llaves:

```
{
  proposición1;
  proposición2;
  .
  .
  proposiciónn;
}
```

Esto da origen a una proposición compuesta o bloque. No hace falta un punto y coma al final de la llave.
Dado un bloque, podemos hacer que el programa repita su ejecución varias veces, o lo ejecute solo si se da una condición dada, etc.

Condicional

En caso de querer ejecutar un bloque de código solo si una condición se cumple en el estado actual, podemos utilizar la construcción `if`:

```
if (condición)
  proposición
```

Condicional

En caso de querer ejecutar un bloque de código solo si una condición se cumple en el estado actual, podemos utilizar la construcción `if`:

```
if (condición)
  proposición
```

Por ejemplo, verificamos si `x` es negativo, de serlo, le asignamos a `sig` el valor -1:

```
if (x < 0)
  sig = -1;
```

Condicional - `else`

Existe una forma más completa de condicional, que sirve para especificar código en caso de que no se cumpla la condición:

```
if (condición)
  proposición
else
  proposición
```

Condicional - `else`

Existe una forma más completa de condicional, que sirve para especificar código en caso de que no se cumpla la condición:

```
if (condición)
  proposición
else
  proposición
```

Por ejemplo, el siguiente código guarda en `abs` el valor absoluto de `x`:

```
if (x >= 0)
  abs = x;
else
  abs = -x;
```

Repetición - `while`

Si queremos repetir una proposición o bloque de instrucciones mientras se cumpla una condición, podemos utilizar el comando `while`:

```
while (condición)
 proposición
```

Repetición - `while`

Si queremos repetir una proposición o bloque de instrucciones mientras se cumpla una condición, podemos utilizar el comando `while`:

```
while (condición)
 proposición
```

Por ejemplo, el siguiente programa decrementa `x` mientras sea positiva:

```
while (x > 0) {
  printf("x : %d\n", x);
  x = x - 1;
}
```

Repetición - `for`

Una forma más compleja de repetición es la proposición `for`. Su forma es:

```
for (expr1; expr2; expr3)
  proposición
```

`expr1` se ejecuta una vez antes de entrar al ciclo.
`expr2` es una condición que controla el ciclo, si es verdadera ejecuta el cuerpo, ejecuta `expr3` y vuelve a evaluarse la condición. El ciclo termina cuando la condición se vuelve falsa.

Repetición - `for`

Una forma más compleja de repetición es la proposición `for`. Su forma es:

```
for (expr1; expr2; expr3)
  proposición
```

`expr1` se ejecuta una vez antes de entrar al ciclo.
`expr2` es una condición que controla el ciclo, si es verdadera ejecuta el cuerpo, ejecuta `expr3` y vuelve a evaluarse la condición. El ciclo termina cuando la condición se vuelve falsa.

Si se omite `expr1` o `expr3`, solo se desechan. Si se omite `expr2`, se toma como permanentemente verdadera.

Condicional - for - ejemplo

```
#include <stdio.h>

int main(void) {
  int i;
  for (i = 1; i < 5; i++) {
    printf("valor i : %d\n", i);
  }
}
```

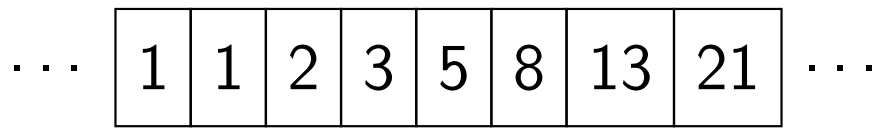
¿Qué hace este programa?

Ejercicios

- Escribir `for` en términos de `while`.
- Escribir `while` en términos de `for`.
- Escribir un ciclo que no termine, que imprima “hola mundo” indefinidamente.

Arreglos

Un arreglo es una secuencia de datos de algún tipo específico. Por ej., un arreglo de 8 enteros sería visto en la memoria como:



Cada uno de estos elementos se encuentra en una celda contigua a la otra, por lo que sus direcciones de memoria son continuadas[2].

[2] Esto es una simplificación.

Constantes simbólicas

Es una mala práctica dejar "constantes mágicas" que no proporcionan demasiada información a quien tenga que leer el programa.
Para evitar esto, se usan constantes simbólicas. Estas se definen usando `#define`:

```
#define nombre texto
```

donde `nombre` es el nombre de la constante, y `texto` es por lo que se reemplazará cuando se encuentre ese nombre en el resto del programa.

Constantes simbólicas

Es una mala práctica dejar "constantes mágicas" que no proporcionan demasiada información a quien tenga que leer el programa.
Para evitar esto, se usan constantes simbólicas. Estas se definen usando `#define`:

```
#define nombre texto
```

donde `nombre` es el nombre de la constante, y `texto` es por lo que se reemplazará cuando se encuentre ese nombre en el resto del programa.
Por ejemplo:

```
#define VALOR_ESPERADO 42
```

Ejemplo - Programa conversión

```
#include <stdio.h>
#define LOWER 0
#define UPPER 300
#define STEP 200

int main(void) {
  int fahr, celsius;
  fahr = LOWER;

  while (fahr <= UPPER) {
    celsius = 5 * (fahr-32) / 9;
    printf("%d\t%d\n", fahr, celsius);
    fahr = fahr + STEP;
  }
}
```

Arreglos - uso

Un arreglo se declara con la sintaxis:

```
tipo nombre[elementos];
```

Arreglos - uso

Un arreglo se declara con la sintaxis:

```
tipo nombre[elementos];
```

Por ej.:

```
int notas[3];
```

Arreglos - uso

Un arreglo se declara con la sintaxis:

```
tipo nombre[elementos];
```

Por ej.:

```
int notas[3];
```

Para acceder a sus elementos se utiliza la notación de corchetes:

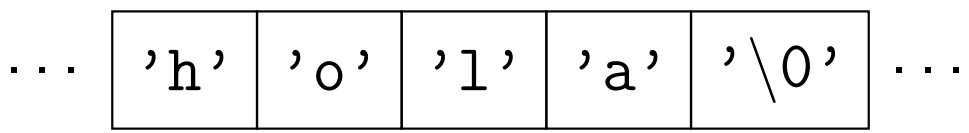
```
notas[0] = 9;
notas[1] = 10;
notas[2] = 3;
prom = (notas[0]+notas[1]+notas[2])/3;
```

Arreglos de carácteres

Podemos representar cadenas utilizando arreglos de caracteres. Por ej., la palabra "hola" podemos pensarla compuesta de cuatro caracteres: `'h'`, `'o'`, `'l'`, `'a'`.

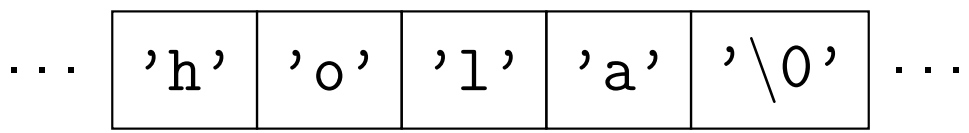
Arreglos de carácteres

Podemos representar cadenas utilizando arreglos de caracteres.
Por ej., la palabra "hola" podemos pensarla compuesta de cuatro caracteres: `'h'`, `'o'`, `'l'`, `'a'`.
Además, utilizaremos un carácter especial extra (`'\0'`) para marcar el final de la cadena.
En memoria, esta cadena se ve de la siguiente manera:



Arreglos de carácteres

Podemos representar cadenas utilizando arreglos de caracteres.
Por ej., la palabra "hola" podemos pensarla compuesta de cuatro caracteres: `'h'`, `'o'`, `'l'`, `'a'`.
Además, utilizaremos un carácter especial extra (`'\0'`) para marcar el final de la cadena.
En memoria, esta cadena se ve de la siguiente manera:



Sabiendo que terminan con `'\0'`, no es necesario saber el tamaño de la cadena, podemos reconocer su final.

Arreglos - inicializadores

Se pueden inicializar los arreglos con una notación compacta que permite definir sus elementos:

```
int nros1[3] = { 1, 2, 3 };
int nros2[] = { 1, 2, 3 };
char texto1[] = {'h', 'o', 'l', 'a', '\0'};
char texto2[] = "hola";
```

Declaración de funciones

En lugar de disponer todo el código dentro de la rutina `main`, utilizaremos funciones para abstraer partes del mismo. La declaración de una función tiene la forma:

```
tipo nombre(tipo arg1, ..., tipo argn);
```

Declaración de funciones

En lugar de disponer todo el código dentro de la rutina `main`, utilizaremos funciones para abstraer partes del mismo. La declaración de una función tiene la forma:

```
tipo nombre(tipo arg1, ..., tipo argn);
```

Por ejemplo, una función de nombre `pot` que toma un flotante `base` y un entero `exp` y devuelve un flotante se declara de la siguiente manera:

```
float pot(float base, int exp);
```

`main` es simplemente una función más: devuelve un entero y no toma argumentos.

Definición de funciones

Para especificar e comportamiento debemos definirla:

```
tipo nombre(tipo arg1, ..., tipo argn) {
  declaraciones
  proposiciones
}
```

Si queremos declarar una función que funcione como un procedimiento (i.e. no retorne valor), usamos el tipo `void` como tipo de retorno.
Si queremos una función que no tome ningún argumento, usamos escribimos `void` donde van los argumentos.

Definición de funciones - ejemplo

Por ejemplo, la función `pot` quedaría definida de la siguiente forma:

```
float pot(float base, int exp) {
  float res = 1;
  while (exp > 0) {
    res = base * res;
    exp--;
  }
  return res;
}
```

Llamando funciones

Podemos llamar a una función previamente declarada utilizando el nombre de la misma, y pasándole entre paréntesis los parámetros correspondientes. Por ej.:

```
int main(void) {
  float res;
  res = pot(2, 3);
  printf("%f", res);
  return 0;
}
```

Entrada/Salida - pantalla (`putchar`)

La primitiva más básica para imprimir en pantalla es `putchar`:

```
int putchar(int c);
```

La función toma un carácter (representado con un entero), y devuelve el carácter que imprimió en la pantalla.

Entrada/Salida - pantalla (putchar)

La primitiva más básica para imprimir en pantalla es `putchar`:

```
int putchar(int c);
```

La función toma un carácter (representado con un entero), y devuelve el carácter que imprimió en la pantalla.
Por ej.:

```
putchar('a');
putchar('l');
putchar('o');
```

imprimirá "alo" en la pantalla. Aquí estamos omitiendo el valor de retorno, i.e. tirándolo.

Entrada/Salida - teclado (`getchar`)

La primitiva más básica para tomar datos del teclado es `getchar`:

```
int getchar(void);
```

La función devuelve el próximo carácter a ser leído disponible.

Entrada/Salida - teclado (getchar)

La primitiva más básica para tomar datos del teclado es `getchar`:

```
int getchar(void);
```

La función devuelve el próximo carácter a ser leído disponible. Por ej.:

```
c = getchar();
b = getchar();
putchar(b);
putchar(c);
```

tomará dos caracteres del teclado y los imprimirá en orden inverso.

Bibliografía

- Kernighan & Ritchie, El lenguaje de programación C.
- The GNU C Reference Manual.
- Wikilibro sobre C.