Programación C

Con estructuras y punteros

E. Rivas

Marzo de 2015

Entrada/Salida

Utilizaremos el término Entrada/Salida (Input/Output, o I/O) para referirnos a las interacciones de nuestro programa con el entorno:

- Pantalla.
- Teclado.
- Archivos del sistema.
- Conexión de red.

En particular, nos concentraremos en la pantalla y teclado.

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. Por ej.:

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

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

Calificadores

Además de los tipos dados, podemos modificar los tipos o variables con diferentes opciones:

- `unsigned`: por cada tipo entero (int, long, etc.), podemos existe uno sin signo ("todos positivos").
- `const`: modifica una variable para especificar que no puede ser cambiado su valor.

Calificadores

Además de los tipos dados, podemos modificar los tipos o variables con diferentes opciones:

- `unsigned`: por cada tipo entero (int, long, etc.), podemos existe uno sin signo ("todos positivos").
- `const`: modifica una variable para especificar que no puede ser cambiado su valor.

Por ej.:

```
unsigned int nat;
const int magic = 42;
```

Creación de tipos de datos

Podemos renombrar tipos de datos utilizando la construcción `typedef`. Su forma es:

```
typedef tipo nombre;
```

A partir de ese momento, el nombre elegido podrá utilizarse como un alias del tipo dado al mismo.

Creación de tipos de datos

Podemos renombrar tipos de datos utilizando la construcción `typedef`. Su forma es:

```
typedef tipo nombre;
```

A partir de ese momento, el nombre elegido podrá utilizarse como un alias del tipo dado al mismo.
Por ej.

```
typedef unsigned int NATURAL;
```

`NATURAL` podrá usarse como un alias de `unsigned int`.

Enumeraciones

Usando enumeraciones podemos crear “nuevos tipos de datos”, cuyos valores serán alguno de los elegidos por nosotros. La sintaxis general es:

```
enum nombre { elemento1, ... elementon };
```

Enumeraciones

Usando enumeraciones podemos crear "nuevos tipos de datos", cuyos valores serán alguno de los elegidos por nosotros. La sintaxis general es:

```
enum nombre { elemento1, ... elementon };
```

Por ej., podemos construir un tipo de datos para valores booleanos:

```
enum booleano { FALSO, VERDADERO };
```

A los valores posibles de una enumeración se los llama elementos o constantes de enumeración.

Enumeraciones - uso

Una vez declarada una enumeración, se pueden definir variables de ese tipo:

```
enum booleano { FALSO, VERDADERO };
enum booleano p = FALSO, q = VERDADERO;
q = FALSO;
```

Enumeraciones - internamente

Internamente, las enumeraciones son simplemente números enteros que se encuentran en biyección con los elementos definidos.
Por defecto, asociará al primer elemento con el 0, al segundo con el 1, etc.

Enumeraciones - internamente

Internamente, las enumeraciones son simplemente números enteros que se encuentran en biyección con los elementos definidos.
Por defecto, asociará al primer elemento con el `0`, al segundo con el `1`, etc.
Se puede especificar el valor a asociar:

```
enum color { VERDE = 1, ROJO, AZUL = 42 };
```

Asocia a `VERDE` con `1`, a `ROJO` con `2`, y a `AZUL` con `42`.

Estructuras

Un punto, en la geometría analítica está definido por dos componentes, su abscisa y su ordenada.
Es decir, un punto queda definido por dos datos: dos números flotantes.

Estructuras

Un punto, en la geometría analítica está definido por dos componentes, su abscisa y su ordenada.
Es decir, un punto queda definido por dos datos: dos números flotantes.
En C, podemos construir un tipo de datos que está compuesto por componentes de otros tipos de datos. Su sintaxis es:

```
struct nombre {
  declaración1;
  .
  .
  declaraciónn;
};
```

donde cada una de las declaraciones define los componentes de la estructura.

Estructuras - ejemplo

En el caso particular del punto, podemos introducir la estructura que define puntos en el plano:

```
struct point {
  float x;
  float y;
};
```

Estructuras - ejemplo

En el caso particular del punto, podemos introducir la estructura que define puntos en el plano:

```
struct point {
  float x;
  float y;
};
```

Para declarar elementos, y acceder a sus campos:

```
struct point p1;
p1.x = 4.5f;
p1.y = 3.4f;
printf("Coordenadas: %f %f", p1.x, p1.y);
```

En este punto `p1` representa el punto $(4.5, 3.4)$.

Estructuras - ejercicio

- A partir de dos puntos se puede definir un rectángulo: escribir una función que tome dos puntos y calcule la superficie del rectángulo correspondiente.
- Defina una estructura que almacene la información correspondiente un registro de agenda: deberá llevar un nombre (arreglo de caracteres de 20 elementos) y un entero sin signo que representa el teléfono de la persona.

Conversión

En muchos casos, necesitamos convertir entre diferentes tipos de datos.
Existen dos tipos de conversión:

Conversión

En muchos casos, necesitamos convertir entre diferentes tipos de datos.
Existen dos tipos de conversión:

- Explícita: de manera explícita se convierte un valor de un tipo en otro.

Conversión

En muchos casos, necesitamos convertir entre diferentes tipos de datos.
Existen dos tipos de conversión:

- Explícita: de manera explícita se convierte un valor de un tipo en otro.
- Implícita: por el uso de operadores se fuerza una conversión.

Conversión - explícita

La conversión explícita se realiza agregando frente a una expresión el tipo de datos nuevo al que queremos hacer la conversión.

Conversión - explícita

La conversión explícita se realiza agregando frente a una expresión el tipo de datos nuevo al que queremos hacer la conversión.
Por ej., la siguiente expresión convierte una expresión flotante a entero:

```
(int)(3.4+f)
```

asumiendo que `f` es un flotante.
Esta expresión tiene tipo entero, y se trunca la parte decimal.

Conversión - implícita

La conversión implícita se realiza cuando por los operadores utilizados se fuerza una conversión de tipo de datos.

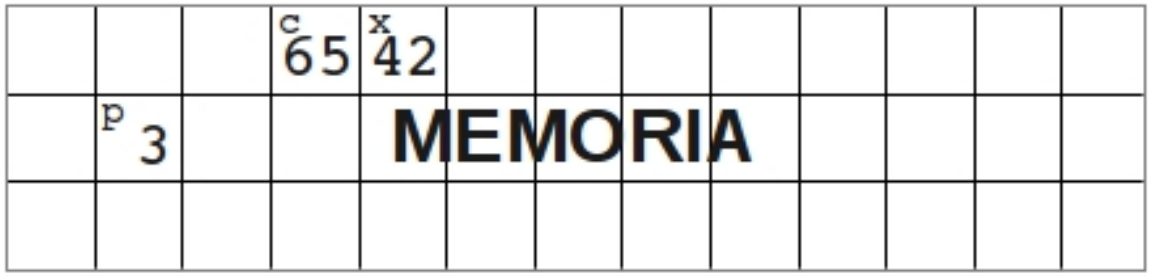
Conversión - implícita

La conversión implícita se realiza cuando por los operadores utilizados se fuerza una conversión de tipo de datos.
Por ej., si `i` es de tipo `int` y `f` es de tipo `float`, entonces la expresión

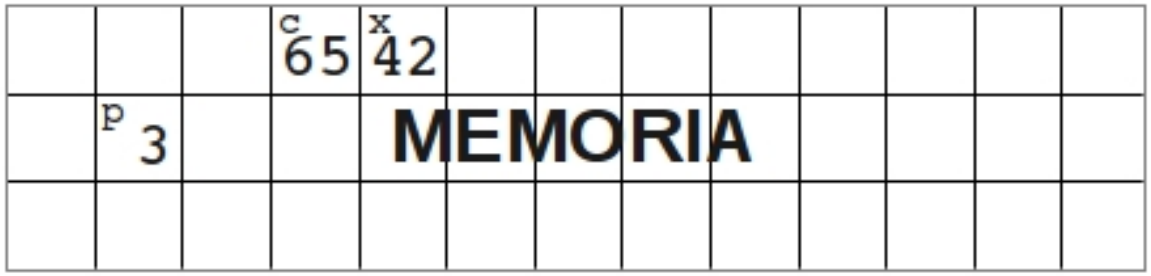
```
i + f
```

promueve `i` a un `float` de manera de poder sumar ambos números.

Punteros - un ejemplo



Punteros - un ejemplo




```
char c = 'A';
int x = 42;
printf("celda: %p\n", &c);
```

Punteros

Un puntero es una variable que contiene la dirección de una variable.
Cada puntero puede contener la dirección de una variable de un tipo particular.

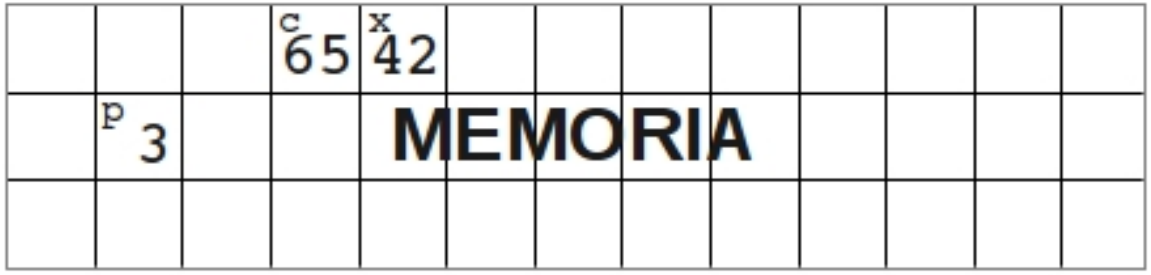
Punteros

Un puntero es una variable que contiene la dirección de una variable.
Cada puntero puede contener la dirección de una variable de un tipo particular.
Se define utilizando un `*` junto al nombre de la variable:

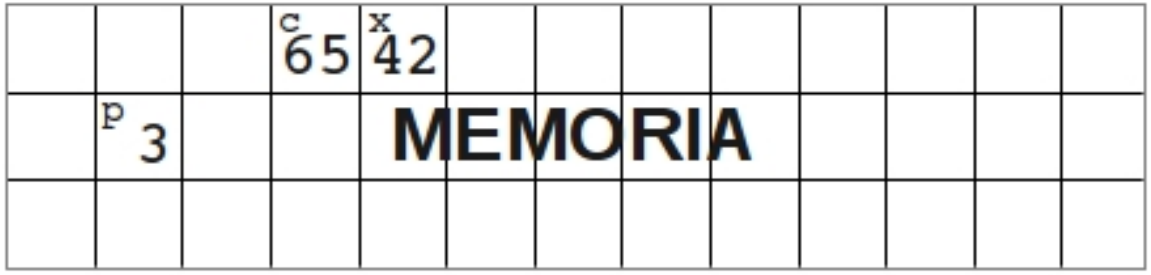
```
tipo *nombre;
```

Si queremos almacenar una dirección de un tipo cualquiera, podemos utilizar el tipo de dato `void *` ("puntero a `void`").

Punteros - otro ejemplo



Punteros - otro ejemplo




```
char c = 'A';
char *p;
p = &c;
*p = 'B';
```

Punteros - arreglos

Un arreglo es una región continua de memoria de un tipo de dato dado. Si tenemos la dirección del primer dato del arreglo, podemos acceder a los otros elementos.

Punteros - arreglos

Un arreglo es una región continua de memoria de un tipo de dato dado. Si tenemos la dirección del primer dato del arreglo, podemos acceder a los otros elementos.

```
int lst[] = { 1, 3, 5 };
int *ptr = &lst[0];
printf("%d\n", *ptr);
printf("%d\n", *(ptr+1));
```

Funciones y punteros

Una función puede recibir un puntero.

```
void settres(int *p) {
  *p = 4;
}
int main(void) {
  int x = 3;
  settres(&x);
  printf("%d\n", x);
  return 0;
}
```

Punteros a funciones

Así como podemos tener variables que contienen la dirección de memoria de una variable, también podemos tener variables que contienen la dirección de memoria de una función.

Punteros a funciones

Así como podemos tener variables que contienen la dirección de memoria de una variable, también podemos tener variables que contienen la dirección de memoria de una función.

```
int (*p)(int);
p = &factorial;
printf("%d", p(3));
```

Manejo dinámico de memoria

Para las variables globales y automáticas se reserva espacio de manera automática. Pero podemos reservar un pedazo de memoria de manera manual.

Manejo dinámico de memoria

Para las variables globales y automáticas se reserva espacio de manera automática. Pero podemos reservar un pedazo de memoria de manera manual.
C provee una función `malloc` (`stdlib.h`), que toma la cantidad de bytes a reservar en memoria, y devuelve la dirección dónde se ha reservado el bloque.

Manejo dinámico de memoria

Para las variables globales y automáticas se reserva espacio de manera automática. Pero podemos reservar un pedazo de memoria de manera manual.
C provee una función `malloc` (`stdlib.h`), que toma la cantidad de bytes a reservar en memoria, y devuelve la dirección dónde se ha reservado el bloque.

```
char *ptr;
ptr = malloc(100);
```

Ahora `ptr` almacena la dirección de un bloque de 100 bytes reservados en memoria.

Liberación de memoria - `free`

Toda memoria pedida con `malloc` queda alocada para nosotros de por vida. Una vez que no utilicemos más una región de memoria pedida, debemos "devolverla" al sistema.

Liberación de memoria - `free`

Toda memoria pedida con `malloc` queda alocada para nosotros de por vida. Una vez que no utilicemos más una región de memoria pedida, debemos "devolverla" al sistema. Esto se realiza usando la función `free`, que permite liberar memoria reservada previamente.

Liberación de memoria - `free`

Toda memoria pedida con `malloc` queda alocada para nosotros de por vida. Una vez que no utilicemos más una región de memoria pedida, debemos "devolverla" al sistema. Esto se realiza usando la función `free`, que permite liberar memoria reservada previamente.

```
char *ptr;
ptr = malloc(100);
.
.
free(ptr);
```

Leer datos - `scanf`

Podemos leer datos (enteros por ej.) utilizando la función `scanf`.

Leer datos - `scanf`

Podemos leer datos (enteros por ej.) utilizando la función `scanf`.
Su primer argumento es una cadena de formato (como en `printf`), y luego la dirección de los elementos donde almacenar los valores leídos.

Leer datos - `scanf`

Podemos leer datos (enteros por ej.) utilizando la función `scanf`.
Su primer argumento es una cadena de formato (como en `printf`), y luego la dirección de los elementos donde almacenar los valores leídos.

```
int x;
printf("Ingrese un nro.: ");
scanf("%d", &x);
```