El Lenguaje de programación C

Sistemas Operativos

Universidade da Coruña

El Lenguaje de programación C I

El Lenguaje de programación C II

Control de flujo

Funciones y estructura de un programa

El Lenguaje de programación C III

- funciones
- funciones que no devuelven enteros
- Variables externas
- Variables estáticas
- Variables registro
- Inicialización
- Recursividad
- EL preprocesador C
- Ejercicios

Arrays y punteros

- Punteros y direcciones
- Punteros y argumentos a funciones
- Operaciones sobre punteros
- Arrays y punteros
- Arrays de punteros
- Arrays multidimensionales

El Lenguaje de programación C IV

El Lenguaje de programación C V

- Documentación en línea
- Asignación de memoria
- Funciones de caracteres
- Cadenas de caracteres
- Entrada/salida con formato
- Entrada/salida con formato a fichero
- Entrada/salida con formato a memoria
- Entrada/salida sin formato
- Ejercicios

Herramientas

- Valgrind

Introducción

Características del lenguaje C

- Es el lenguaje de programación de propósito general asociado al sistema operativo UNIX
- Es un lenguaje de medio nivel. Trata con objetos básicos como carácteres, números . . . ; también con bits y direcciones de memoria
- Posee una gran portabilidad
- Se utiliza para la programación de sistemas: construcción de interpretes, compiladores, editores de texto, etc

- El lenguaje C consta de
  - El lenguaje C propiamente dicho: tipos de datos, expresiones y estructuras de control
  - Extensiones en forma de macros y un amplio conjunto de librerías predefinidas

Introducción

Estructura de un programa C

- Un programa e C consta de uno o más módulos (ficheros fuentes)
- Cada módulo puede contener
  - directivas del precompilador, p.e para "incluir" otros ficheros (**#include**) y "definir" constantes y macros (**#define**)
  - declaraciones de variables y prototipos de funciones
  - una o más funciones
  - comentarios
- Cada función puede contener
  - directivas del precompilador
  - declaraciones
  - uno o más bloques
  - comentarios

- Cada bloque puede contener
  - directivas del precompilador
  - declaraciones
  - una o más sentencias
  - comentarios

- Cada sentencia debe estar terminada por `;`
- Cada bloque de sentencias se encierra entre llaves $\{\ldots\}$
- La función denominada `main` es la que primero se ejecuta
- Los comentarios pueden aparecer en cualquier lugar del código y se insertan entre `/*` y `*/` así

  ```
  /* esto es un comentario*/
  ```

- o entre `//` y final de línea

  ```
  // esto es otro comentario
  ```

Introducción

Primeros ejemplos

```
#include <stdio.h>
main()
{
  printf("hola, primer programa en C\n");
}
```

```
#include <stdio.h>
main()
{
  int fahr, celsius;
  int lower, upper, step;
  lower=0;
  upper=300;
  step=20;
  fahr=lower;
  while(fahr<=upper)
  {
    celsius=5*(fahr-32)/9;
    printf("%d\t%d\n",fahr,celsius);
    fahr=fahr+step;
  }
}
```

Introducción

Compilación de un programa

- Normalmente invocamos al compilador con `cc` o `gcc` (el compilador de gnu)
- Cuando tecleamos `cc programa.c` para generar un ejecutable a partir de un fichero fuente obtenemos un ejecutable, típicamente denominado `a.out`. Sin embargo se realizan tres tareas
  - Paso por el preprocesador C. Es el que procesa las líneas que comienzan con # (pe **#include**, **#define**...). Puede invocarse directamente con `cpp`
  - La compilación propiamente dicha. Genera ficheros objeto (`.o`) a partir de los ficheros fuente. Puede invocarse como `cc -c`
  - El enlazado. Realizado por `ld`

Introducción

Directivas del precompilador

#include Incluye el fichero, cuyo nombre se indica, para su compilación con el resto del código

- Si el nombre del fichero va entre `<...>` se busca en el directorio include del sistema (`/usr/include`). Por ejemplo

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

  hace que el fichero `/usr/include/stdio.h` se incluya en ese punto para ser compilado con nuestro fichero fuente
- El nombre se puede poner entre comillas (“) y el sistema lo buscará en el directorio actual (también se puede poner una ruta absoluta)
- Los ficheros se suelen denominar `.h` y contienen declaraciones de tipos, variables y prototipos de funciones (son ficheros fuente que pueden verse)

#define Define un símbolo, es decir cada ocurrencia de ese símbolo es sustituida por su definición. Por ejemplo

- El C distingue entre mayúsculas y minúsculas, aunque la costumbre es utilizar mayúsculas para los símbolos definidos por el preprocesador
- ejemplo

  ```
  #define MAX 1024
  ```

  hace que todas las ocurrencias de la cadena MAX dentro del código son sustituidas por 1024
- El código que se compila lleva 1024 donde iba MAX
- Si MAX iba dentro de comillas ’’...‘‘, no se sustituye

```
#include <stdio.h>
#define UPPER 300
#define LOWER 0
#define STEP 20
main()
{
  int fahr, celsius;
  fahr=LOWER;

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

Introducción

Depuración

- Para depurar programas disponemos del depurador, que nos permite ejecutar los programas en un entorno controlado
- Para usar el depurador
  a. Compilamos con `cc -g`

     `$gcc -g ejercicio1.c`
  b. Invocamos el depurador pasándole el ejecutable como parámetro

```
antonio@abyecto:~$ gdb a.out
GNU gdb (GDB) 7.3-debian
Copyright (C) 2011 Free Software Foundation, Inc.
.......
(gdb)
```

  - Dentro del depurador podemos establecer puntos de ruptura con `break` (`break numerolínea` o `break nombrefunción`), ejecutar paso a paso con `step` o `next`, ver variables con `display` .... El depurador dispone de ayuda en línea, mediante `help`

Introducción

make

- Facilita el proceso de generación y actualización de un programa.
- Determina automáticamente qué partes de un programa deben recompilarse ante una actualización de algunos módulos y las recompila.
- Se crean un archivo `Makefile` con las dependencias entre los ficheros: un fichero debe actualizarse si alguna de sus dependencias es más reciente
- El `Makefile` define también los mandatos necesarios para actualizar cada archivo en base a sus dependencias

- El `Makefile` esta formado por una serie de reglas. Donde cada regla tiene el siguiente formato
  objetivo: dependencia$_1$ dependencia$_2$ ...
  mandato$_1$
  mandato$_2$
  mandato$_3$
  ...
- Nótese que antes de cada mandato hay un caracter *tab*
- Para compilar usamos la orden ‘‘`make objetivo`’’ que
  - Encuentra la regla correspondiente a *objetivo*
  - Trata sus dependencias como objetivos y las resuelve recursivamente
- Dentro del `Makefile` se pueden definir variables (y acceder a sus valores) como en el *shell*: VAR=valor para definir la variable y ${VAR} o $(VAR) para acceder a su valor. $@ se refiere al nombre del objetivo

Ejemplo muy sencillo de make

- Consideremos el siguiente programa (`programa.c`)

```
#include "funcion1.h"
#include "funcion2.h"

main()
{
  int a=MAX1,b=MAX2;
  funcion1();
  funcion2();
}
```

- Donde `funcion1.h`

```
#define MAX1 1000
void funcion1(void);
```

- y `funcion2.h`

```
#define MAX2 2000
void funcion2(void);
```

Ejemplo muy sencillo de make

- Además, `funcion1.c` es

```
#include "funcion1.h"

void funcion1(void)
{
  int i=MAX1;
  return;
}
```

- y `funcion2.c` es

```
#include "funcion2.h"

void funcion2(void)
{
  int j=MAX2;
  return;
}
```

Ejemplo muy sencillo de make

- si queremos compilarlo, y generar un programa llamado `programa.out`
  1. Compilaríamos `funcion1.c`

     ```
     gcc -c  funcion1.o funcion1.c
     ```
  2. Compilaríamos `funcion2.c`

     ```
     gcc -c  funcion2.o funcion2.c
     ```
  3. Compilaríamos `programa.c` de la siguiente manera

     ```
     gcc  -o programa.out programa.c funcion2.o funcion1.o
     ```
- Cada vez que se modificase uno de los archivos habría que repetir alguna (o todas) de las anteriores compilaciones

Ejemplo muy sencillo de make

- El siguiente archivo `Makefile` se ocupa de decidir que hay que compilar y hacerlo

```
programa.out: programa.c funcion1.h funcion2.h funcion1.o funcion2.o
	gcc  -o programa.out programa.c funcion2.o funcion1.o

funcion1.o: funcion1.h funcion1.c
	gcc -c  funcion1.o funcion1.c

funcion2.o: funcion2.h funcion2.c
	gcc -c  funcion2.o funcion2.c

limpiar:
	rm programa.out funcion1.o funcion2.o
```

Otro ejemplo de make

```
CC=gcc
CFLAGS=-g
OBJS2=prac2.o aux1.o

all: prac1 prac2

prac1: prac1.o aux1.o
	gcc -g -o prac1 prac1.o aux1.o

prac2: $(OBJS2)
	${CC} ${CFLAGS} -o $@ ${OBJS2}

prac1.o prac2.o: prac.h

clean:
	rm -f prac1.o aux1.o ${OBJS2}
```

Introducción

Ejercicios

- Compilar los programas de ejemplo de esta sección
- Observar las salida del preprocesador de cada uno de ellos
- Usar el depurador para ejecutar paso a paso los dos últimos y ver como cambian de valor las variables en el bucle
- Crear un `Makefile` para un archivo y compilarlo con make

Tipos, operadores y expresiones

Nombres de variables

- los nombres de variables pueden tener letras, números y el símbolo '_'.
- Deben empezar por una letra (pueden empezar por '_' pero no es recomendable pues es el criterio que usan las rutinas de la biblioteca)
- Pueden llevar mayúsculas y minúsculas. El C distingue entre mayúsculas y minúsculas
  - La costumbre es que las variables van en minúscula y las constantes en mayúscula
- Las palabras reservadas *if*, *else* ... no pueden usarse como nombres de variables

Tipos, operadores y expresiones

Tipos y tamaños de datos

- El C tiene pocos tipos de datos

char	Un byte. Contiene un caracter (o un número entre 0 y 255)
int	Un entero. El tamaño depende del sistema donde estemos
float	Un real
double	Un real en doble precisión

- Además *int* puede ser *short* o *long* y tanto *int* como *char* pueden ser *signed* o *unsigned*
  - unsigned long int, unsinged long,
- El tamaño depende del compilador pero *int* no es menor que *short* ni mayor que *long*
- Existe también el *long long int* con mayor rango y el *long double* con precisión extendida para los reales

Tipos, operadores y expresiones

Constantes

- Una constante entera es *int* (ejemplo 12714)
- Si queremos que sea *long* le ponemos el sufijo 'l' o 'L'. (ejemplo 12714L)
- Una constante con un punto decimal o un exponente es un *double* (ejemplos 3.141516, 1e-5)
- Si queremos que sea *float* le ponemos el sufijo 'f' o 'F' (ejemplos 3.141516f, 1e-5F)
- Si una constante entera comienza por 0 se entiende que está en octal y si comienza por 0x o 0X en hexadecimal
  - Ejemplos 034892 (octal) 0xffff4000 (hexadecimal)

- Una constante carácter se pone entre comillas simples (ejemplo 'a'). Puede aparecer en operaciones y su valor es el código de dicho caracter
- Una constante carácter también se puede expresar por su código en octal o hexadecimal. Ejemplos: '\077' (octal) o '\x3f' (hexadecimal)
- Algunas constantes especiales: '\t' (tab) '\n' (fin de línea') '\r' retorno de carro ...
- Una constante cadena se encierra entre comillas dobles ("... "). Las cadenas en C están terminadas por el caracter '\0',

- Una expresión constante es una expresión que solo incluye constantes. Puede ser evaluada en tiempo de compilación y ser utilizada en lugar de una constante

```
#define MAXIMO 1024
char linea[MAXIMO/2 +20];
```

- Un caso particular de constantes son los tipos enumerados. Un tipo enumerado es una lista de valores enteros constantes

```
enum boolean {FALSE, TRUE};
```

- Salvo que se especifique otra cosa, comienzan en 0, así FALSE sería 0 y TRUE 1
- También puede especificarse el valor de comienzo

```
enum dias {LUNES=1, MARTES, MIERCOLES,
      JUEVES, VIERNES, SABADO, DOMINGO};
```

Tipos, operadores y expresiones

Declaraciones de variables

- Todas las variables deben ser declaradas antes de su uso.
- Las declaraciones deben aparecer al principio de cada función o bloque de sentencias
- La declaración consta de un tipo de variable y una lista de variables separadas por coma

```
int i,j;
float x,pi;
unsigned long longitud, contador;
```

- Las variables pueden inicializarse en la declaración

```
float pi=3.1416;
unsigned long contador=0;
```

- Puede utilizarse el simbol *const* para indicar que la variable no puede ser cambiada

```
const float e=2.7182;
```

Tipos, operadores y expresiones

Operadores aritméticos

- `+` suma
- `-` resta
- `*` multiplicación
- `/` división. Si los operandos son enteros la división es entera
- `%` resto entero. No admite operandos *float* o *double*

```
#include <stdio.h>
#define UPPER 300
#define LOWER 0
#define STEP 20
main()
{
  float fahr, celsius;
  fahr=LOWER;

  while(fahr<=UPPER)
  {
    celsius=5.0/9.0*(fahr-32)
    printf("%f\t%f\n",fahr,celsius);
    fahr=fahr+STEP;
  }
}
```

Tipos, operadores y expresiones

Operadores de relación y lógicos

- Operadores de relación
  - $<$ menor que
  - $<=$ menor o igual que
  - $>$ mayor que
  - $>=$ mayor o igual que
  - $==$ igual a
  - $!=$ distinto

- Operadores lógicos

  && AND

  || OR

  ! NOT

- Los operadores lógicos se evalúan de izquierda a derecha, y se detiene la evaluación tan pronto como se conoce el resultado. Por ejemplo

  ```
  i<MAXIMO && (linea[i]=getchar())!='\n' && linea[i]!=EOF
  ```

  Comprobaría primero que i es menor que MAXIMO; si lo es asigna el valor que devuelve *getchar()* a linea[i] y comprueba que es distinto de *fin de línea*. y en ese caso comprueba que linea[i] no es EOF

Tipos, operadores y expresiones

Conversiones de tipo

- Cuando en una operación aparecen tipos distintos, el C convierte el tipo de menor rango al tipo de mayor antes de realizar la operación
- Podemos forzar la conversión de tipos poniendo antes de la variable el tipo al que queremos convertir.
- Por ejemplo, la función *sin* espera un argumento de tipo *double*. Podemos pasarle una variable entera haciendo una conversión de tipo

```
double seno; int n;
seno= sin ((double) n);
...
double centigrados, fahrenheit=55.3;
centigrados = 5/9 * (fahrenheit - 32); //77F = 25C
centigrados = 5.0/9 * (fahrenheit - 32);
```

Tipos, operadores y expresiones

Operadores de bit

- Podemos operar sobre los bits de una variable entera (las variables char son un tipo de enteros). Todos son operadores binarios, excepto el complemento a 1
  - `&` AND bit a bit
  - `|` OR bit a bit
  - `^` XOR bit a bit
  - `<<` Desplazamiento de bits a la izquierda
  - `>>` Desplazamiento de bits a la derecha
  - `~` Complemento a 1

Tipos, operadores y expresiones

Incremento y decremento, asignación y expresiones

- Los operadores incremento ($++$) y decremento ($--$) pueden usarse dentro de expresiones. Si preceden a la variable, el incremento o decremento se produce antes de usarla, en caso contrario después. Ejemplo:

  `linea[i++]='a';`

  asigna a linea[i] el valor 'a' y luego incrementa el valor de i, mientras que

  `linea[--i]='z'`

  primero decrementa el valor de i y luego le asigna a linea[i] el valor 'z'
- Cuando se modifica el valor de una variable a partir de un valor anterior, puede escribirse de forma más compacta
- xOPERADOR =expresión; equivale a x=x OPERADOR (expresión);

x+=expresión; equivale a x=x+expresión;
x-=expresión; equivale a x=x-expresión;
x*=expresión; equivale a x=x*expresión;
...
x>> =expresión; equivale a x=x>>expresión;
x&=expresión; equivale a x=x&expresión;
...

```
x+=2; /*x=x+2*/
x*=4+y;  /*x=x*(4+y)*/
```

Tipos, operadores y expresiones

Expresión condicional

- Es de la forma $expre_1 ? expre_2 : expre_3$
- Se evalúan $expre_1$, si es cierta (distinta de 0) el resultado es lo que valga $expre_2$
- Si es 0, el resultado es lo que valga $expre_3$
- Ejemplos

```
char * a;
   a= (p==NULL)? "cadena1": gets(p);

int n;
   n= i*j*k ? 2567: (int) sqrt (pi);

printf ("Resultado: %s\n" (p!=NULL)? p:" error");
```

Tipos, operadores y expresiones

Precedencia y orden e evaluación

- El C no especifica en que orden se evalúan los operandos de una expresión. Por ejemplo, en la expresión $x = f() + g()$ no sabemos si se evaluará primero f() o g()
- El C tampoco especifica en que orden se evalúan los parámetros a una función,

  `printf (‘‘%d %d’’,++i,i)`

  produce un resultado indeterminado
- Las operaciones lógicas se evalúan de izquierda a derecha y se detiene la evaluación tan pronto como se conozca el resultado
- El C sí define una precedencia entre distintos operadores, Además cada operador tiene una asociatividad, como puede verse en la tabla siguiente

OPERADORES	asociatividad
`() [] -> .`	izquierda a derecha
`! ~ ++ -- - * &` (tipo) `sizeof`	derecha a izquierda
`* /`	izquierda a derecha
`+ -`	izquierda a derecha
`<< >>`	izquierda a derecha
`< <= > >=`	izquierda a derecha
`== !=`	izquierda a derecha
`&`	izquierda a derecha
`^`	izquierda a derecha
`|`	izquierda a derecha
`&&`	izquierda a derecha
`||`	izquierda a derecha
`?:`	derecha a izquierda
`= += -= *= /= %= &= |=` etc	derecha a izquierda
`,`	izquierda a derecha

Tipos, operadores y expresiones

Ejercicios

- Escribir un programa en C que nos imprima los tamaños de todos los tipos de datos
- Escribir una función *InvierteBitsC* que invierte los bits de un caracter sin signo
- Escribir una función *InvierteBitsS* que invierte los bits de un entero corto sin signo.
- Escribir una función *InvierteBitsL* que invierte los bits de un entero largo sin signo
- Hacer un programa que imprima los enteros del 0 al 100 y el resultado de invertir los bits
  - Considerando los números del 0 al 100 como caracteres sin signo
  - Considerando los números del 0 al 100 como enteros cortos sin signo
  - Considerando los números del 0 al 100 como enteros largos sin signo

Control de flujo

Sentencias y bloques

`if else`

`else-if`

`switch`

`bucleswhile, for` y `do..while`

`break` y `continue`

`goto` y `etiquetas`

Ejercicios

- Una expresión seguida en un **;** es una sentencia
  - Toda sentencia debe ir terminada con **;**
  - Pueden ir varias sentencias en una misma línea
- Las llaves { } se emplean para agrupar sentencias. en lo que se denomina *bloque*
- Un bloque es sintácticamente equivalente a una sentencia
- Dentro de un bloque puede haber sentencias y declaraciones

Control de flujo

if else

if (expresión)
  sentencia$_1$
else
  sentencia$_2$

- Se evalúa *expresión*. Si es distinta de 0 (cierta) se ejecuta sentencia$_1$, si es 0 se ejecuta sentencia$_2$
- El **else** es opcional
- Tanto sentencia$_1$ como sentencia$_2$ pueden ser un bloque se sentencias entre llaves { }
- Expresión no termina con **;**

- Dado que el **else** es opcional, una sentencia **else** siempre va con el **if** inmediatamente anterior
- El siguiente código es incorrecto

```
if (n>0)
  if (a>b)
    printf ("a es mayor que b\n");
else
  printf ("n es menor que 0\n");
```

- La versión correcta sería

```
if (n>0) {
  if (a>b)
    printf ("a es mayor que b\n");
}
else
  printf ("n es menor que 0\n");
```

Control de flujo

else-if

if (expresión$_1$)
  sentencia$_1$
else if(expresión$_2$)
  sentencia$_2$
else if(expresión$_3$)
  sentencia$_3$
...
else
  sentencia$_n$

- Permite tomar decisiones múltiples
- Se evalúan en orden

Control de flujo

switch

switch (expresión) {
  case constante$_1$**:** sentencias
  case constante$_2$**:** sentencias
  case constante$_3$**:** sentencias
  ...
  default: sentencias
}

- Permite tomar una decisión múltiple basada en una expresión que puede tomar un número de valores constantes enteros (los *char* son un tipo de entero)
- El valor de expresión se va comparando con las constantes por orden y una vez que coincide con una de ellas se ejecutan todas las sentencias (incluidas las de los *cases* siguientes) hasta que se encuentra la sentencia **break**
  - Esto permite agrupar varios valores constantes que compartan una misma acción
  - Si no queremos que pase de un *case* al siguiente, debemos terminarlo con **break**

```
#include <stdio.h>
main() /* count digits, white space, others */
{
  int c, i, nwhite, nother, ndigit[10];
  nwhite = nother = 0;
  for (i = 0; i < 10; i++)
    ndigit[i] = 0;
  while ((c = getchar()) != EOF) {
    switch (c) {
      case '0': case '1': case '2': case '3': case '4':
      case '5': case '6': case '7': case '8': case '9':
         ndigit[c-'0']++;
         break;
      case ' ':
      case '\n':
      case '\t':
         nwhite++;
         break;
      default:
         nother++;
         break;
    }
  }
  printf("digits =");
  for (i = 0; i < 10; i++)
     printf(" %d", ndigit[i]);
  printf(", white space = %d, other = %d\n", nwhite, nother);
return 0;
}
```

Control de flujo

bucles`while`, `for` y `do..while`

while (expresión)
  sentencia

- Se evalúa expresión, y si es distinta de 0 se ejecuta sentencia
- *sentencia* debe estar terminada por **;**
- *sentencia* puede ser un bloque se sentencias entre llaves { }

```
int strlen(char s[])
{
  int i;

  i=0;
  while (s[i]!= '\0')
    ++i;
  return i;
}
```

for ($expr_1$; $expr_2$; $expr_3$)
  sentencia

- Equivale exactamente a

$expr_1$;
while ($expr_2$){
  sentencia
  $expr_3$;
}

- La expresiones del **for**, a diferencia de otros lenguajes, no tienen por que ser referidas a enteros, pueden ser expresiones de cualquier tipo

```
int atoi(char s[])
{
  int i, n;

  n=0;
  for(i=0; s[i]>='0' && s[i]<='9'; ++i)
    n=10*n+(s[i]-'0');
  return n;
}
```

do
  sentencia
while (expresión)

- Se ejecuta sentencia y después se evalúa expresión, con lo que *sentencia* se ejecuta **al menos** una vez
- *sentencia* debe estar terminada por **;**
- *sentencia* puede ser un bloque se sentencias entre llaves { }

Control de flujo

break y continue

- C proporciona dos modos de salida de los bucles: **break** y **continue**

break Provoca la salida del bucle. Si hay varios bucles anidados provoca la salida de aquel donde se encuentra

continue Provoca la salida de la presente iteración del bucle. Se vuelve a la condición.

- **break** se utiliza también para la salida del **switch**

```
int i;
for (i=1;i<10;i++) {
   if (i)
       continue;  //break;
   printf("\n i vale %d",i);
}
```

Control de flujo

goto y etiquetas

- Aunque no son formalmente necesarias, ni convenientes, C dispone de una sentencia **goto** y etiquetas
- Una etiqueta tiene el mismo formato que un nombre de variable, seguida de dos puntos
- Debe estar en la misma función en donde se encuentra el goto

```
for ( ... )
   for ( ... ) {
     for ( ... ) {
         ...
         if (disaster)
           goto error;
     }
  ...
error:
  /* clean up the mess */
```

Control de flujo

Sentencias y bloques
`if else`
`else-if`
`switch`
`bucleswhile, for` y `do..while`
`break` y `continue`
`goto` y `etiquetas`
Ejercicios

Ejercicios

- Escribir una función en C *ConvierteAEnt*, que a partir de una cadena de caracteres que contiene la representación de un entero en una base, y dicha base nos devuelve el entero
- Escribir una función en C *ConvierteACad*, que a partir de un entero y una base nos devuelve una representación de entero en dicha base
  - Comprobar que son correctas
  - Podemos suponer que la longitud máxima de la cadena es 16
- hacer un programa en C que a imprima todos los enteros del 32 al 64 en todas las bases de 2 a 16. Imprime un entero (en las 16 bases) por línea

Funciones y estructura de un programa

funciones

- Un programa e C es una colección de funciones.
- Una de esas funciones se llama *main* y es la primera en ejecutarse
- Las funciones pueden residir en uno o varios ficheros fuente
- Cada función tiene la forma
  tipo_de_dato nombre_funcion (declaraciones_argumentos)
  {
  declaraciones y sentencias
  }
- Se puede omitir el tipo de dato que devuelve la función (en ese caso se asume que es **int**

- Una función puede no tener argumentos o no tener declaraciones o sentencias

```
funcion_simple()
{
}
```

- Cuando una función no lleva parámetros o no devuelve ningún valor se usa el término *void*

```
void funcion_nada(void)
```

```
#include <stdio.h>

#define MAXLINE 1000 /* maximum input line length */

int getline(char line[], int max)
int strindex(char source[], char searchfor[]);

char pattern[] = "ould"; /* pattern to search for */

/* find all lines matching pattern */
main()
{
  char line[MAXLINE];
  int found = 0;

  while (getline(line, MAXLINE) > 0)
    if (strindex(line, pattern) >= 0) {
```

```
      printf("%s", line);
      found++;
    }
  return found;
}

/* getline: get line into s, return length */
int getline(char s[], int lim)
{
  int c, i;
  i = 0;

  while (--lim > 0 && (c=getchar()) != EOF && c != '\n')
    s[i++] = c;
  if (c == '\n')
    s[i++] = c;
  s[i] = '\0';
  return i;
```

```
}

/* strindex: return index of t in s, -1 if none */
int strindex(char s[], char t[])
{
  int i, j, k;

  for (i = 0; s[i] != '\0'; i++) {
    for (j=i, k=0; t[k]!='\0' && s[j]==t[k]; j++, k++)
      ;
    if (k > 0 && t[k] == '\0')
      return i;
    }
  return -1;
}
```

- Las funciones devuelven valores mediante la sentencia **return**
  - El formato es
    return *expresión*;
    o
    return *(expresión)*;
  - *expresión* se convierte al tipo de dato que devuelve la función
  - Puede aparecer en cualquier parte de una función y provoca la inmediata salida de ella
  - Si dicha sentencia aparece dentro de un bucle provoca la salida de la función y por tanto del bucle

Funciones y estructura de un programa

funciones que no devuelven enteros

- C asume que toda función, salvo que se especifique en contrario, devuelve un entero
- Si tenemos una función que devuelve un valor que no es un entero, y sobre todo, si la definición de dicha función aparece en el código después de la primera vez que es llamada, debemos declararla
- Esto se hace declarando el tipo que devuelve la función.
- Podemos declarar también los parámetros que recibe, así el compilador podrá comprobar la sintaxis cuando es llamada
- En el ejemplo siguiente podemos ver que se declaran los valores devueltos por las funciones llamadas desde *main*

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

#define MAXLINE 100
/* rudimentary calculator */
main()
{
  double sum, atof(char []); //<-- Declaración función
  char line[MAXLINE];
  int getline(char line[], int max);

  sum = 0;
  while (getline(line, MAXLINE) > 0)
    printf("\t%g\n", sum += atof(line));
  return 0;
}
```

```
double atof(char s[])
{ double val, power;   int i, sign;

  for (i = 0; isspace(s[i]); i++) /* skip white space */
     ;
  sign = (s[i] == '-') ? -1 : 1;
  if (s[i] == '+' || s[i] == '-')
    i++;
  for (val = 0.0; isdigit(s[i]); i++)
    val = 10.0 * val + (s[i] - '0');
  if (s[i] == '.')
    i++;
  for (power = 1.0; isdigit(s[i]); i++) {
    val = 10.0 * val + (s[i] - '0');
    power *= 10;
  }
  return sign * val / power;
}
```

Funciones y estructura de un programa

Variables externas

- Variables externas son las que están definidas fuera del cuerpo de cualquier función
- Son compartidas por todos las funciones en mismo fichero fuente desde donde está definida la variable hasta el final del fichero
- En C, las variables definidas dentro de bloques se llaman *automáticas*, se crean automáticamente cuando comienza la ejecución del bloque y se desasignan al terminar dicha ejecución

- El dominio (zona donde se reconoce el nombre) de las variables automáticas se reduce al bloque donde están declaradas
- El dominio de los argumentos a una función es dicha función
- Si en un bloque interior se declara una variable con el mismo nombre que una en un bloque mas exterior (o una externa) dentro de dicho bloque el nombre se refiere a la declarada en él
- Si queremos que una variable externa sea compartida por varios ficheros fuente distintos, debe definirse en uno de ellos y en los otros declararla con el identificador *extern*

Funciones y estructura de un programa

Variables estáticas

- Una variable estática es una variable local de una función que no se crea y destruye al llamar a la función
- Se declaran anteponiendo la palabra *static* a la declaración

```
void funcioncilla (void)
{
  static int veces=0;

  ++veces
  printf ("Esta funcion ha sido llamada %d\n" veces);
}
```

- Desde el punto de vista del almacenamiento, una variable estática es en realidad una variable externa, y el termino *static* la hace privada a la función
- Si una variable externa (o una función) la declaramos *static* la hacemos privada para el fichero fuente donde está definida

Funciones y estructura de un programa

Variables registro

- La declaración *register* antes del nombre de una variable indica al compilador que esa variable va a usarse intensamente y que seria conveniente almacenarla en un registro de la máquina

  ```
  register int indice;
  ```

- El compilador tiene libertad de colocarla en un registro o no
- Solo es aplicable a variables automáticas y parámetros de las funciones
- No puede preguntarse por la dirección de una variable registro, aunque de hecho no se almacene en un registro

Funciones y estructura de un programa

Inicialización

- C permite que las variables sean inicializadas en la declaración
- Las variables externas y estáticas se inicializan una sola vez durante la compilación
- Las inicializaciones de variables automáticas son sentencias de asignación (se ejecutan)
- Las expresiones de inicialización deben ser conocidas en el momento de la inicialización: Las variables externas y estáticas solo pueden inicializarse a valores constantes
- Los arrays también pueden inicializarse mediante una lista de sus elementos entre llaves y separados por ,

```
int dias_por_mes[] = { 31, 28, 31, 30, 31, 30,
                       31, 31, 30, 31, 30, 31 }
```

- En este caso no es necesario declarar la dimensión del array

Funciones y estructura de un programa

Recursividad

- C permite recursividad en las funciones, tanto directa como indirecta
- No es necesaria ninguna declaración especial

```
#include <stdio.h>
/* printd: print n in decimal */
void printd(int n)
{
  if (n < 0) {
    putchar('-');
    n = -n;
  }
  if (n / 10)
    printd(n / 10);
  putchar(n % 10 + '0');
}
```

Funciones y estructura de un programa

EL preprocesador C

- Además de la directiva **#include** y el uso elemental del **#define**, el preprocesador C tiene otras funcionalidades
- Uso de **#define** para definir macros

```
#define CUBO(x)  x*x*x   /*incorrecto CUBO (x+2)
#define CUBO(x) (x)*(x)*(x)
#define MAX(a,b) ((a>b)?a:b)
#define MAX(a,b,R) if (a > b) R=a; else R=b;
```

- Con **#ifdef** e **#ifndef** podemos saber si un símbolo ha sido definido o no. Ejemplo: prevenir que un fichero *include* se incluya varias veces, (lo que daría símbolos duplicados)

```
#ifndef _UNISTD_H
#define _UNISTD_H
.....
#endif
```

- Puede dejarse sin efecto una definición

```
#undef MAXIMO
```

- Puede comprobarse si se han definido símbolos con algún valor concreto, para hacer construcciones más complejas

```
#if SYSTEM == SYSV
  #define HDR "sysv.h"
#elif SYSTEM == BSD
  #define HDR "bsd.h"
#elif SYSTEM == MSDOS
  #define HDR "msdos.h"
#else
  #define HDR "default.h"
#endif

#include HDR
```

Funciones y estructura de un programa

Ejercicios

- Repartir los programas de ejemplo de este tema en varios ficheros fuente (uno para cada función), de manera que se puedan compilar separadamente. Hacer los ficheros include correspondientes
- Declarar una variable externa con el mismo nombre en cada fichero fuente y comprobar si es o no la misma
- Declarar todas las variables locales como register y compilarlo
- Hacer que la variable externa declarada en el apartado anterior sea la misma para todos los ficheros
- Hacer en C una función recursiva que calcula en factorial de un número. Imprimir los 20 primeros factoriales

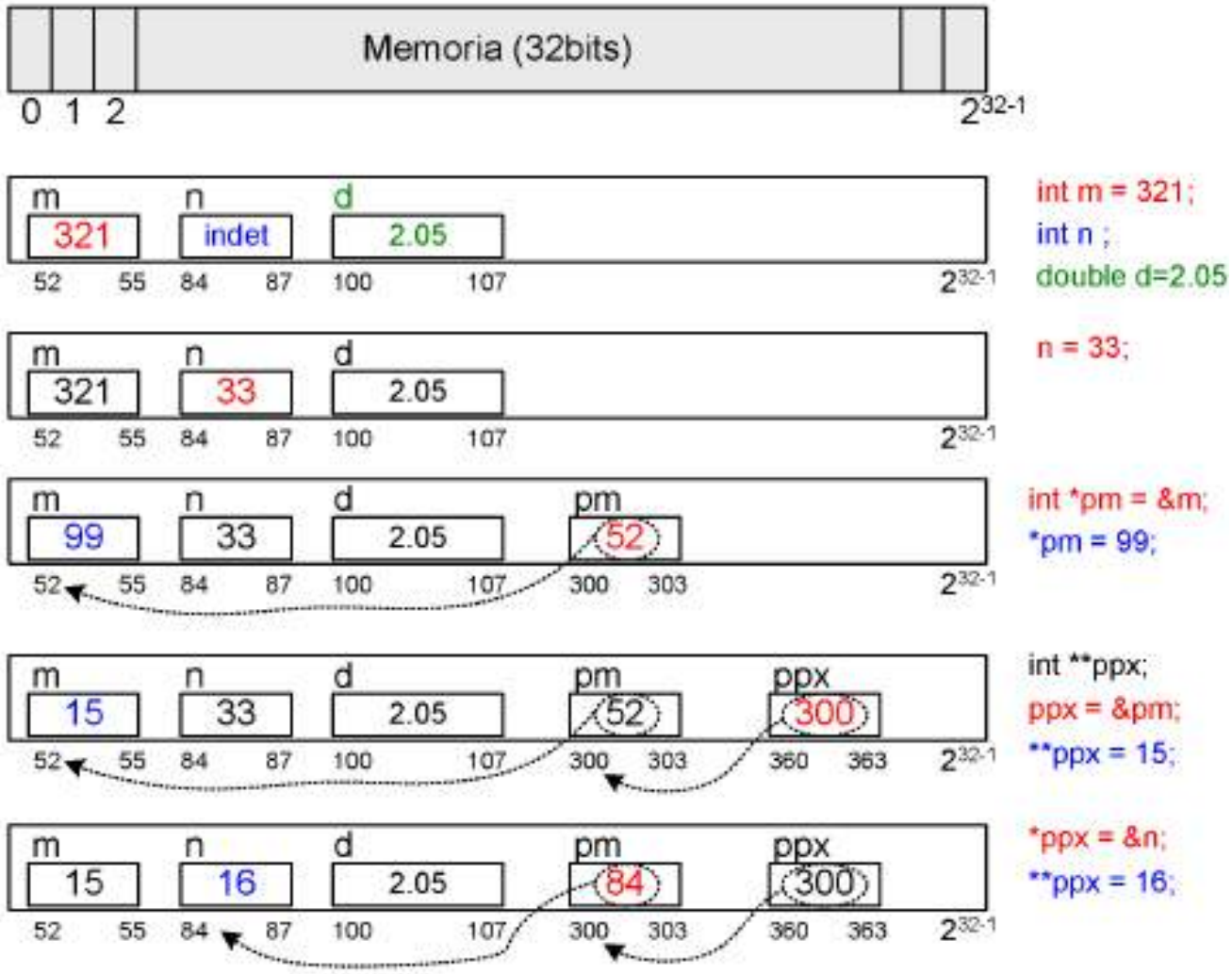
Arrays y punteros

- Un puntero es una variable que contiene la dirección de un dato. C proporciona dos operadores relacionados con las direcciones de memoria
  - `*` Operador indirección. A partir de una variable tipo puntero nos proporciona el dato apuntado
  - `&` Operador dirección. A partir de una variable nos da la dirección de memoria donde se almacena dicha variable
- Para declarar un puntero se declara el tipo de dato apuntado

```
int *p;
double *pf;
```

declara que `*p` es un entero, por tanto `p` es un puntero a entero
`*pf` es un double, por tanto `pf` es un puntero a un `double`

- Organización de Memoria: Ejemplo



- La declaración del puntero reserva memoria para la variable puntero **NO PARA EL OBJETO APUNTADO**. En el ejemplo anterior se reserva memoria para `p`, El acceso a `*p` tiene un resultado indefinido, pudiendo resultar en un error en tiempo de ejecución
- Antes de usar un puntero debemos asegurarnos que apunta un una dirección correcta
  - Asignándole la dirección de una variable. Por ejemplo

    ```
    p=&i;
    ```
  - Asignándole el valor que devuelva una función que reserve memoria para él. Por ejemplo

    ```
    p=(int *) malloc (sizeof (int));
    ```
  - Tras utilizar el puntero, siempre se debe liberar la memoria reservada con `malloc` utilizando la función `free`

    ```
    free(p);
    ```

Arrays y punteros

Punteros y argumentos a funciones

- en C todas las llamadas a función son por valor: La función recibe siempre una copia de lo que se le pasa como argumento. Las modificaciones se realizan sobre la copia
- Las siguiente función no afecta a los argumentos que se le pasan

```
void intercambia (int x, int y) /* INCORRECTA */
{
  int temp;

  temp = x;
  x = y;
  y = temp;
}
```

Una llamada a `intercambia (a,b)` no intercambiaría los valores de `a` y `b`, solo de sus copias

- Si queremos que la función sea llamada por referencia, lo que hacemos es que la función reciba las direcciones de las variables que queremos modificar, y a través de ellas acceda a las variables

```
void intercambia(int *px, int *py)
{
  int temp;

  temp = *px;
  *px = *py;
  *py = temp;
}
```

- Para intercambiar dos variables `a` y `b` la llamaríamos

```
intercambia(&a,&b);
```

Arrays y punteros

Operaciones sobre punteros

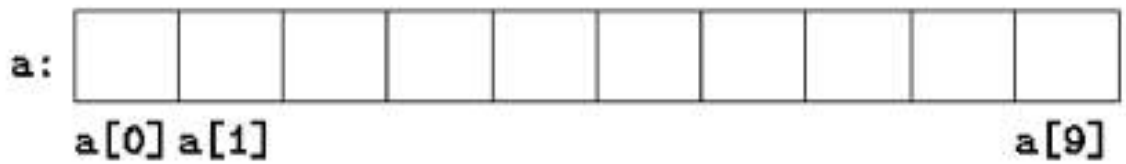
- C permite las siguientes operaciones sobre punteros
  - **puntero + entero** La suma (y la resta), tienen en cuenta el tamaño del objeto apuntado de manera que si `p` apunta a un entero, `p+1` apuntaría al siguiente entero
  - **puntero - entero**
  - **puntero = puntero** Asignación entre punteros
  - **puntero = NULL**
  - **puntero == NULL** Comparación con NULL
  - **puntero != NULL**
  - **puntero ==puntero** La comparación entre punteros tiene restricciones

Arrays y punteros

Arrays y punteros

- La declaración en C de un array es
  `tipo nombre_array[dimension]`
- La siguiente declaración declara un array de 10 enteros

  `int a [10];`
- Los elementos se acceden como a[0], a[1] . . . a[9]
- Los elementos de un array se almacenan consecutivamente en memoria

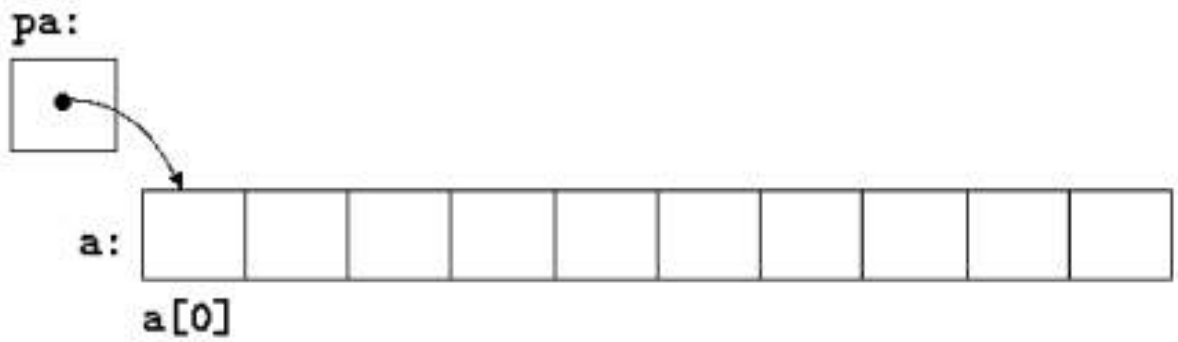


- El nombre del array es la dirección del primer elemento del array

,

- Consideremos ahora lo siguiente

```
int *pa;
pa=&a[0];
```



- los elementos del array `a[0], a[1]...a[9]` están almacenados en las direcciones de memoria `pa,pa+1,...pa+9` por lo que pueden ser accedidos como *pa, *(pa+1),...*(pa+9)
- Dado que el nombre del array es la dirección del primer elemento del array, podríamos haber hecho `pa=a` en lugar de `pa=&a[0];`

,

- C también admite el acceso a los elementos del array de esta manera

  `pa[0], pa[1]  pa[9]`

- Hay que tener en cuenta que, aunque el nombre del array es la dirección del primer elemento del array, no es una variable, sino una constante, por lo que una sentencia del tipo

  `a=pb;`

  producirá un error en tiempo de compilación

,

- Si queremos pasar un subarray a una función podemos hacerlo de manera muy sencilla. Con las declaraciones anteriores

  ```
  f(a+3);
  f(&a[3]);
  ```

  pasarían un subarray comenzando en el tercer elemento del array.
- La declaración de los parámetros en la función que recibe el array puede ser (suponiendo que no devuelve nada)

  ```
  void f (int ar[]) { ....}
  ```

  o

  ```
  void f (int *ar) { ....}
  ```

```c
#include <stdio.h>
char a[2][5] = {{11,12,13,14,15},
                {21,22,23,24,25}};

main() {
  char *p;       // puntero a char
  char (*q)[5]; // puntero a un array de 5 chars
  char *r[5];   // array de 5 punteros a char

  p = &(a[0][0]);

  printf("%d\n", p[0]);
  p++;
  printf("%d\n", p[0]);

  q = &(a[0]);

  printf("%d == %d \n", q[0][0], (*q)[0]);  // equivalentes

  q++;

  printf("%d\n", q[0][0]);

  p = *a;

  printf("a = %ld \t *a = %ld \t **a = %ld \n", a, *a, **a);
  printf("p = %ld \t *p = %ld \n", p, *p);
}
```

Arrays y punteros

Arrays de punteros

- En C un puntero es como cualquier otra variable (en concreto es un tipo de entero)
- Pueden hacerse arrays de punteros.
- La declaración

  `int * ar[MAX]`

  declara que `ar` es un array de MAX punteros a enteros. Los elementos `ar[i]` son punteros. El acceso a `*ar[i]`, mientras no inicialicemos adecuadamente los `ar[i]` produce un resultado indefinido
- De la misma manera que en el apartado anterior existía una relación muy estrecha entre el puntero y el array, aquí el puntero equivalente al array `ar` se declararía

  `int **par;`

Arrays y punteros

Arrays multidimensionales

- C proporciona arrays multidimensionales, aunque se usan más los arrays de punteros.
- La declaración de un array de dos dimensiones es

  `tipo nombre_array[dim1][dim2]`

  ```
  int matriz [FILAS][COLUMNAS];
  ```

- El acceso al elemento `i,j` es `matriz[i][j]`

```
static char daytab[2][13] = {
  {0, 31, 28, 31, 30, 31, 30, 31, 31, 30, 31, 30, 31},
  {0, 31, 29, 31, 30, 31, 30, 31, 31, 30, 31, 30, 31}
 };

/* day_of_year: set day of year from month & day */

int day_of_year(int year, int month, int day)
{
  int i, leap;

  leap = year%4 == 0 && year%100 != 0 || year%400 == 0;
  for (i = 1; i < month; i++)
    day += daytab[leap][i];

  return day;
}
```

```
/* month_day: set month, day from day of year */

void month_day(int year, int yearday, int *pmonth,
                                      int *pday)
{
  int i, leap;

  leap = year%4 == 0 && year%100 != 0 || year%400 == 0;
  for (i = 1; yearday > daytab[leap][i]; i++)
    yearday -= daytab[leap][i];

  *pmonth = i;
  *pday = yearday;
}
```

Arrays y punteros

Punteros y arrays multidimensionales

- Aunque en C existen los arrays multidimensionales, no suelen usarse pues es mas conveniente usar los arrays de punteros. La declaración

  ```
  int a[10][20];
  ```

  declara un array de 10x20 enteros (10 filas y 20 columnas).
  - **a** es la dirección de un bloque donde hay 200 enteros. Para acceder al elemento a[1][3] tengo que saber que cada fila tiene 20 columnas
  - Si quiero pasar dicho array a una función tendría que pasarle la segunda dimensión ara que pudiese acceder correctamente a los enteros

    ```
    int func (int arr[][20])
    ```

- Consideremos ahora esta otra declaración

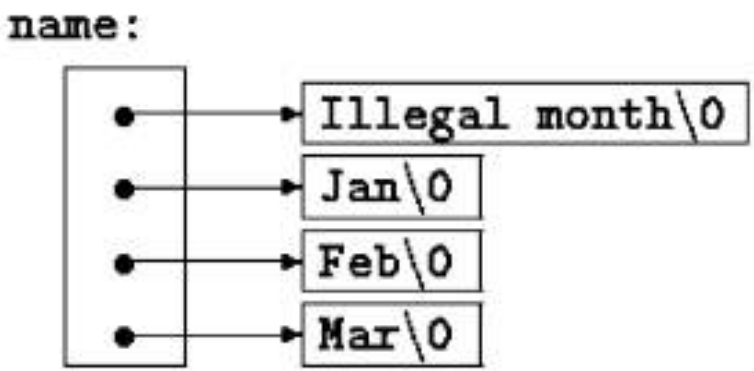
  ```
  int * p[10];
  ```

  declara un array de 10 punteros a entero

- Si a cada uno de los punteros le asigno memoria para 20 enteros, tengo de nuevo una matriz de 10x20 enteros a la que puedo acceder como `p[i][j]`.
- En memoria tengo 10 bloques de 20 enteros cada uno (no necesariamente consecutivos) (mas 10 punteros)
- El acceso es más rápido pues no necesito multiplicaciones para determinar a donde accedo (solo sumas e indirecciones)
- A una función no tendría que pasarle la segunda dimensión para que pudiese acceder a los enteros
- Es mas flexible, pues no todos los bloques tienen que ser del mismo tamaño. Esto es especialmente interesante cuando queremos cadenas de caracteres

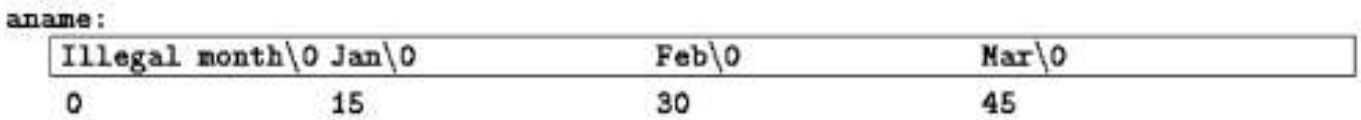
- En las siguientes figuras vemos la diferencia en la disposición de memoria
  - Array de punteros

```
char *name[] = { "Illegal month", "Jan", "Feb", "Mar" };
```



  - Array multidimensional

```
char aname[][15] = { "Illegal month", "Jan", "Feb", "Mar" };
```

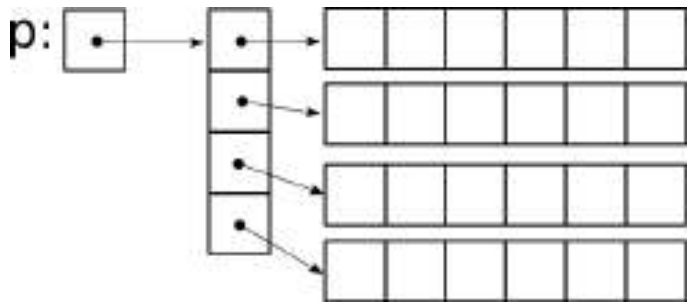


- Con esta otra declaración

```
int **p;
```

creamos un array multidimensional basado en punteros.

- Si queremos crear una matriz de 4 filas por 6 columnas, debemos reservar memoria para filas y columnas




```
p = (int **) malloc (4 * sizeof(int *));
for (i = 0; i < 4; i++)
{
  p[i] = (int *) malloc (6 * size(int));
}
```

- Acceso a elementos de la matriz

```
for (i = 0; i < 4; i++)
{
    for (j = 0; j < 6; j++)
    {
        p[i][j] = 0;          // Equivalentes
        *(p[i] + j) = 0;
        *(*(p + i) + j) = 0;
    }
}
```

- Para liberar la matriz

```
for (i = 0; i < 4; i++)
    free(p[i]);

free(p);
```

Arrays y punteros

Argumentos en la línea de comando

- C proporciona un método para acceder a los argumentos de la línea de comando, a través de los parámetros de *main*

1. Declaramos que *main* tiene dos parámetros; uno un entero y otro un array de punteros a carácter, de la siguiente manera

```
int main(int argv, char argv[])
```

2. Al ejecutar el programa, *argc* tendrá el número de argumentos y los *argv[i]* (hasta *argv[argc-1]*) son los argumentos de línea de comando
   - *argv[0]* es el nombre del programa que se ejecuta por lo que si *argc* es 1, no se le han pasado argumentos

```c
#include <stdio.h>
/* echo command-line arguments; 1st version */

int main(int argc, char *argv[])
{
  int i;

  for (i = 1; i < argc; i++)
    printf("%s%s", argv[i], (i < argc-1) ? " " : "");
  printf("\n");

  return 0;
}
```

Arrays y punteros

Punteros a funciones

- Aunque una función no es una variable, si podemos usar punteros a funciones, de manera que podemos usarlos en arrays o pasarlos como parámetros a otras funciones
- La siguiente declaración

  `double (*pfunc)(int,double)`

  declara que *pfunc* es un puntero a una función que devuelve un *double* y que recibe dos parámetros; un *int* y un *double*

  `char * (*pfunc)(char **)`

  ahora pfunc es un puntero a una función función que devuelve un puntero a caracter y que recibe como argumento un array de punteros a carácter
- El siguiente ejemplo lo ilustramos con la función *qsort* que hace una ordenación de líneas y recibe como parámetro un puntero a la función que compara una línea con otra

```
/* Qsort: sort v[left]...v[right] into increasing order */

void Qsort(void *v[], int left, int right,
           int (*comp)(void *, void *))
{
  int i, last;
  void swap(void *v[], int, int);

  if (left >= right) /* do nothing if array contains */
     return; /* fewer than two elements */
  swap(v, left, (left + right)/2);
  last = left;

  for (i = left+1; i <= right; i++)
    if ((*comp)(v[i], v[left]) < 0)
      swap(v, ++last, i);
```

```
  swap(v, left, last);
  qsort(v, left, last-1, comp);
  qsort(v, last+1, right, comp);
}
```

- Si quisiésemos utilizar la siguiente función para comparar líneas

```
int numcmp(char *s1, char *s2)
{
  double v1, v2;
  v1 = atof(s1); v2 = atof(s2);
  if (v1 < v2)
     return -1;
  else if (v1 > v2)
     return 1;
  else
     return 0;
}
```

teniendo en cuenta que el nombre de la función es un puntero a la función, la llamada sería

```
Qsort (lineas,MAXLINEAS, numcmp);
```

Arrays y punteros

Ejercicios

- Escribir un programa en C que imprime todas las líneas de su entrada estándar que contienen una palabra que se le pasa como argumento
- Escribir un programa en C que admite como parámetro un número entero **N**; si el número es positivo imprime las N primeras líneas de su entrada, y si es negativo las (-)N últimas.
- Escribir un programa en C que ordene todas las líneas que lee de su entrada usando la función *qsort*. Si recibe el parámetro `-n` las ordena por longitud, en caso contrario lo hace alfabéticamente

Estructuras

Estructuras

- Una estructura es una colección de una o mas variables, no necesariamente del mismo tipo, agrupadas bajo un solo nombre
  - Es el equivalente al registro (*record*) del pascal
- La declaración es de la forma
  struct nombre_estructura {lista_variables} nombre_variable **;**
- El nombre de la estructura puede omitirse. Las siguientes declaraciones son (casi) equivalentes: todas declaran dos estructuras (p1 y p2) cada una de las cuales contiene dos enteros

a
```
struct COORDENADAS {
    int x;
    int y;
  } p1, p2;
```

b
```
struct COORDENADAS {
    int x;
    int y;
  };
struct COORDENADAS p1, p2;
```

```c
struct  {
    int x;
    int y;
  } p1 , p2;
```

- En las modalidades a) y b) la estructura tiene un nombre (struct COORDENADAS) que nos podría servir para declarar más estructuras del mismo tipo (o parámetros a una función) en otra parte del programa
- Las estructuras pueden inicializarse en la declaración al igual que otros tipos de variables, Los valores de sus miembros separados por , y entre llaves ({})

```c
struct COORDENADAS p1={ 5,9 };
```

Estructuras

Operaciones sobre estructuras

- C permite las siguientes operaciones sobre estructuras
  - copia o asignación
  - acceso a sus miembros
  - obtener su dirección con &
  - Pasarlas como argumentos a funciones
  - Ser devueltas por funciones
- Las estructuras no se pueden comparar
- Para acceder a los miembros de una estructura usamos el operador .
  Con las declaraciones del apartado anterior

  ```
  p1.x=3
  p1.y=9;
  p2.x=p2.y=0
  ```
- Una estructura puede tener miembros que sean a su vez estructuras

Estructuras

Punteros a estructuras

- C también permite punteros a estructuras.
- La declaración de un puntero a una estructura es como la de un puntero a cualquier otra variable. Se declara el objeto apuntado.
- También son válidas las distintas modalidades vistas

a
```
struct COORDENADAS {
    int x;
    int y;
  } *p1, *p2;
```

b
```
struct COORDENADAS {
    int x;
    int y;
  };
struct COORDENADAS *p1, *p2;
```

```c
struct  {
    int x;
    int y;
  } *p1 , *p2;
```

- Podemos acceder a los miembros de una estructura a través del puntero de dos maneras
  - Accediendo a la estructura

    `(*p1).x`

    Los paréntesis () son necesarios pues el operador `.` tiene precedencia sobre la indirección `*`
  - Directamente desde el puntero mediante el operador `->`

    `p1->x`

    En general este sistema es el preferido, sobre todo cuando usamos variables tipo `struct` para crear estructuras de datos (listas, pilas, árboles . . .)

    `p->siguiente->siguiente->siguiente`
    `(*(*(*p).siguiente).siguiente).siguiente`

- Al igual que con los otros punteros, la declaración de un puntero a una estructura reserva espacio para el puntero pero **NO PARA LA ESTRUCTURA APUNTADA POR ÉL** por lo que el acceso a dicha estructura produce un resultado indefinido (incluso un error en tiempo de ejecución)
- Antes de acceder a la estructura apuntada debemos inicializar el puntero
  - Asignándole la dirección de una variable
  - Reservándole memoria con alguna función (p.e. *malloc*)
- En el siguiente ejemplo reservamos memoria para MAX estructuras `struct COORD` y lo asignamos a un puntero. El operador `sizeof` nos devuelve el tamaño de la estructura
  - El tamaño de una estructura no es necesariamente la suma del tamaño de sus miembros

```
struct CORD *p;
p=(struct CORD *) malloc (MAX * sizeof (struct CORD));
```

Estructuras

Arrays de estructuras

- C permite también hacer arrays de estructuras, la sintaxis es la misma que para cualquier otro tipo de variable
- Los arrays de estructuras también se puede inicializar en la declaración usando llaves para cada estructura y llaves para cada elemento del array

```
struct PARNUMEROS {
  int i;
  double x;
  }  array[]={
  {0, 1000.0},
  {1, 3.14},
  ........
  {100, 2.5}
  };
```

  aunque a veces por comodidad se omiten las llaves de cada estructura
- En el siguiente ejemplo vemos una función que usa un array de estructuras para contar el número de palabras reservadas de C que se utilizan en un fichero

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

#define MAXWORD 100

struct key {
   char *word;
   int count;
   } keytab[] = {
      "auto", 0,
      "break", 0,
      "case", 0,
      "char", 0,
      "const", 0,
      "continue", 0,
      "default", 0,
      /* ... */
      "unsigned", 0,
      "void", 0,
      "volatile", 0,
      "while", 0
   };

int getword(char *, int);
int binsearch(char *, struct key *, int);
```

```
/* count C keywords */
main()
{
    int n;
    char word[MAXWORD];

    while (getword(word, MAXWORD) != EOF)
        if (isalpha(word[0]))
            if ((n = binsearch(word, keytab, NKEYS)) >= 0)
                keytab[n].count++;

    for (n = 0; n < NKEYS; n++)
        if (keytab[n].count > 0)
            printf("%4d %s\n", keytab[n].count, keytab[n].word);

    return 0;
}
```

```
/* binsearch: find word in tab[0]...tab[n-1] */

int binsearch(char *word, struct key tab[], int n)
{
   int cond;
   int low, high, mid;

   low = 0;
   high = n - 1;

   while (low <= high) {
      mid = (low+high) / 2;
      if ((cond = strcmp(word, tab[mid].word)) < 0)
         high = mid - 1;
      else if (cond > 0)
         low = mid + 1;
      else
         return mid;
    }
   return -1;
}
```

Estructuras

Estructuras autoreferenciadas

- Una estructura no puede referenciarse a si misma es decir, tener una estructura del mismo tipo como miembro, puesto que esto daría lugar a una recursión infinita.
- Lo que si puede tener como miembro es uno o varios punteros a una estructura de su mismo tipo
- Esto nos permite realizar estructuras de datos en memoria

```
struct TNODO{
    struct INFO info;
    struct TNODO *izq;
    struct TNODO *der;
};
```

Estructuras

Uniones

- Una unión es una variable que contiene tipos de datos distintos en instantes distintos
- Permite manipular distintos tipos de datos en la misma zona de memoria
- Se declara de manera similar a una estructura pero con la palabra **union**
- El acceso a los miembros de la union es con el operador `.` (o con el operador `->` si accedemos a través de un puntero)

```
union VARIOS {
  int entero;
  char bytes[4];
  float real;
  } u;
```

- Una *union* puede contener arrays y/o estructuras. Una estructura también puede contener uniones. Están permitidos también los arrays de uniones.

Estructuras

Campos de bits

- Permiten, con una variable de tipo **struct** tener acceso directo a los bits de un entero
- Con una declaración de este tipo

```
struct {
    unsigned int is_keyword : 1;
    unsigned int is_extern : 1;
    unsigned int is_static : 1;
} flags;
```

podemos acceder a los bits individualmente como

```
flags.is_keyword=1;
flags.is_extern=0;
..
if (flags.is_static) {..
```

- Si los bits comienzan a asignarse por la izquierda o por la derecha y otros detalles son dependientes de la implementación, por lo que es mas usual usar máscaras y enteros para realizar dichas tareas

```
#define KEYWORD 01
#define EXTRENAL 02
#define STATIC 04
int flags;
flags|=KEYWORD |EXTERN;

if (flags & STATIC) {...
```

Estructuras

typedef

- C proporciona una herramienta para ponerle nombres a tipos: **typedef**
- Su uso es similar a una declaración de variable precedida de la palabra **typedef**. Lo que iría en el lugar del nombre de la variable es el nombre del tipo

```
typedef int * punteroEntero;
typedef struct NUEVA nueva_t;
```

Si ahora quisiéramos declarar una variable puntero a entero y otra de tipo `struct NUEVA` podríamos hacer

```
punteroEntero p;
nueva_t n;
```

Estructuras

Ejercicios

- Implementar e C un programa que lee enteros de su entrada estándar y los imprime en orden inverso, utilizando una pila. Implementar la pila
  - Con un array
  - Con un array de punteros
  - De manera dinámica
- Implementar una lista en C donde cada elemento de la lista contiene una palabra y un entero. El programa lee su entrada (cada línea tiene una palabra y un entero) y almacena los elementos en la lista ordenados por el valor del entero.

Biblioteca C

Biblioteca C

- El lenguaje C es muy pequeño, sin embargo, muchas de las tareas podemos hacerlas por medio de la biblioteca C. por ejemplo, usamos funciones de la biblioteca C para
  - asignar y desasignar memoria
  - manipulado de cadenas y caracteres
  - funciones matemáticas
  - entrada salida
  - ...
- En los sistemas tipo UNIX la información sobre las funciones de la biblioteca C está disponible en la documentación en línea (sección 3 de las páginas de manual). Por ejemplo, para obtener información sobre *printf*

  ```
  $ man 3 printf
  ```

  o en otros sistemas

  ```
  $ man -s 3 printf
  ```

Biblioteca C

Documentación en línea

- La información en línea de una función nos indica, además de la sintaxis de dicha función, si es necesario algún fichero *include* o hay que indicarle alguna librería adicional
- Por ejemplo, si miramos la página de manual de la función `sqrt`, vemos que es necesario incluir el fichero `<math.h>` y que además hay que enlazar con `-lm`

```
SQRT(3)                                          Linux Programmer's Manual

NAME
       sqrt, sqrtf, sqrtl - square root function

SYNOPSIS
       #include <math.h>

       double sqrt(double x);
       float sqrtf(float x);
       long double sqrtl(long double x);

       Link with -lm.
.....
```

Biblioteca C

Asignación de memoria

- Para asignar memoria tenemos

  `void *malloc(size_t size);`

  - Le pasamos como argumento la cantidad de bytes que queremos reservar (típicamente usamos `sizeof` para determinarlo)
  - Nos devuelve una dirección de memoria, como un puntero genérico (`void *`), donde hay disponible dicha cantidad de bytes (no más). Si no queremos que el compilador de un aviso al asignarlo a, por ejemplo, un puntero a entero, debemos hacer una conversión de tipo
- Cuando ya no necesitemos la memoria asignada con *malloc* podemos desasignarla con

  `void free(void *ptr);`
- Utilizar más memoria de la asignada o hacer *free* sobre una dirección que no ha sido obtenida con *malloc* produce resultados indefinidos, típicamente un error en tiempo de ejecución
- Otras funciones relacionadas: calloc, realloc

Biblioteca C

Funciones de caracteres

- La biblioteca C proporciona una serie de funciones que permiten tratar con caracteres. Sus prototipos están declarados e <ctype.h>

```
int isalnum(int c);
int isalpha(int c);
int isascii(int c);
int isblank(int c);
int iscntrl(int c);
int isdigit(int c);
int isgraph(int c);
int islower(int c);
int isprint(int c);
int ispunct(int c);
int isspace(int c);
int isupper(int c);
int isxdigit(int c);
```

Biblioteca C

Cadenas de caracteres

- En C no existen las variable tipo cadenas (*strings*). Existen
  - arrays de caracteres
  - punteros a carácter
- Cuando se almacena una cadena de caracteres debe estar terminada por el carácter cuyo código es 0 (**'\0'**) ya que es lo que esperan encontrar las funciones de cadenas
- Las funciones de cadenas esperan una dirección de memoria (char *); como el nombre de un array es la dirección donde se almacena el array, las funciones de cadenas pueden recibir tanto un array como un puntero
- Hay algunas diferencias sutiles entre el array y el puntero a carácter

- Consideremos la siguiente declaración

  ```
  char a[MAX]="esto es una cadena";
  ```

- Se declara un array de MAX caracteres y se inicializa a ‘‘esto es una cadena’’
- El siguiente código produce un error de compilación

  ```
  a="la cadena ha cambiado de valor";
  ```

  puesto que el nombre del array es la dirección donde comienza el array, y ésta no puede cambiarse
- Podríamos cambiar el valor del array con una de las funciones de cadena

  ```
  strcpy(a,"la cadena ha cambiado de valor");
  ```

- Consideremos la siguiente declaración

  ```
  char *a="esto es una cadena";
  ```

- Se declara un puntero y se inicializa a la dirección de memoria donde está la constante literal ‘‘`esto es una cadena`’’
- El siguiente código NO produce error de compilación ni en tiempo de ejecución

  ```
  a="la cadena ha cambiado de valor";
  ```

  se asigna al puntero la dirección de memoria donde está la constante literal ‘‘`la cadena ha cambiado de valor`’’
- Si intentamos cambiar el valor de puntero con una de las funciones de cadena

  ```
  strcpy(a,"la cadena ha cambiado de valor");
  ```

  el resultado es indefinido puesto que estamos intentando sobreescribir la cadena ‘‘`esto es una cadena`’’ que está donde el compilador ha puesto las constantes literales con otra cadena, que además es de mayor longitud

- Consideremos la siguiente declaración

  ```
  char *a;
  a=(char *) malloc (MAX*sizeof (char));
  ```

  asigna al puntero a una dirección de memoria donde hay espacio para MAX caracteres

- La sentencias

  ```
  strcpy(a,"esto es una cadena");
  strcpy(a,"la cadena ha cambiado de valor");
  ```

  son perfectamente correctas siempre y cuando MAX sea mayor que la longitud de ‘‘`la cadena ha cambiado de valor`’’ +1

- Sin embargo la asignación

  ```
  a="la cadena ha cambiado de valor";
  ```

  hace que el valor del puntero sea ahora el de la dirección de la constante literal ‘‘`la cadena ha cambiado de valor`’’ , y hemos perdido la memoria asignada con *malloc*

- Cuando hacemos una asignación es una asignación entre punteros.
  - Se asignan direcciones de memoria
  - No se copian cadenas
- Si queremos copiar cadenas debemos usar la función de librería *strcpy*
  - Es responsabilidad del programador que en el sitio a donde se copia haya espacio suficiente para la cadena que se quiere copiar
  - El espacio necesario es la longitud de la cadena + 1 byte adicional (para el caracter '\0' que marca el fin de cadena.
- Se muestran posibles implementaciones de las funciones de la librería *strcpy* (que copia una cadena) y *strdup* (que crea un duplicado)

```
void strcpy(char *s, char *t)
{
  int i=0;

  while ((s[i] = t[i]) != '\0')
    i++;
}

char *strdup(char *s) /* make a duplicate of s */
{
  char *p = (char *) malloc(strlen(s)+1);

  if (p == NULL)
     return NULL;
  strcpy(p, s);
  return p;
}
```

- Funciones de caracteres: resumen (string.h)
  - `char *strcpy(char *dest, const char *orig);`
  - `char *strncpy(char *dest, const char *orig, size_t n);`
  - `void *memcpy(void *dest, const void *src, size_t n);`
  - `size_t strlen(const char *s);`
  - `char *strcat(char *dest, const char *src);`
  - `char *strncat(char *dest, const char *src, size_t n);`
  - `int strcmp(const char *s1, const char *s2);`
  - `int strncmp(const char *s1, const char *s2, size_t n);`
  - `strstr`, `strchr`,...
- Otras funciones (stdlib.h): `atoi`, `atof`, `atol`,...

Biblioteca C

Entrada/salida con formato

- El prototipo de estas funciones está en `<stdio.h>`
- Si queremos hacer la entrada salida por caracteres tenemos

```
int getchar()
int putchar(int c)
```

- Para cadenas de caracteres

```
char * gets(char *s)
int puts(const char *s)
```

- La función de entrada salida con formato es *printf(...]*
- El formato de *printf* es

```
int printf(const char *format, ...)
```

format es una cadena de caracteres compuesta por

- caracteres que son copiados a la salida estándar
- especificaciones de conversión: comienzan con le caracter % y terminan con un caracter de conversión.
  - La primera especificación de conversión se refiere al primer argumento después del **format**, la segunda al siguiente ....
  - El significado de los caracteres en una especificación de conversión es:
    - \- El ajuste es a la izquierda
    - **N.M** N ancho mínimo (para *float* dígitos antes del **.** decimal). M número mínimo de dígitos que se imprimirán para un entero (para *float* número de dígitos a la derecha del **.** decimal)
    - **h** para entero corto **l** para entero largo
    - Caracter de conversión

- El caracter de conversión especifica qué hay que imprimir

Especificaciones de conversión (%u, %d,...)			Códigos de escape	
carácter	**argumento**	**salida**	**Código**	**salida**
d,i	entero	entero con signo	\n	nueva línea
u	entero	entero sin signo	\t	tabulador
o	entero	entero en octal sin signo	\b	backspace
x, X	entero	entero en hexadecimal sin signo	\r	retorno carro
f	real	real con punto y signo	\¨	comillas
e,E	real	notación exponencial con signo	\´	apóstrofo
g, G			\\	\
c	carácter	carácter	\?	?
s	cad. Chars	cadena de caracteres		
%		imprime un %		
p	void	depende implementación		
ld, lu, lx, lo	entero	entero largo		

- Por ejemplo:

```
printf (‘‘real: %2.4f; entero: %X\n", x,n);
```

imprimiría algo como:

```
real: 3.1516; entero: FF09465
```

- Si queremos hacer entrada con formato, utilizamos *scanf*

  `int scanf(const char *format, ...);`
- La especificación de los formatos de *scanf* es como la de *printf*.
- Hay que tener en cuenta que en la lista de argumentos a *scanf* no se le suministran las variables que queremos leer, sino `las direcciones de memoria` de las variables que queremos leer (para que sea por referencia)
- Ejemplo

```
int n; char caracter;
scanf ("%d",&n);
printf ("\n El cuadrado de  %d es : %d",n, n*n);
scanf ("%c",&caracter);
printf ("\n He leído el carácter: %c", caracter);
```

Biblioteca C

Entrada/salida con formato a fichero

- Las funciones *printf*, *scanf*, *puts*, *gets*, *putchar*, *getchar* operan sobre la entrada estándar y la salida estándar. Existen funciones totalmente análogas que operan sobre ficheros
- Los ficheros podemos abrirlos con *fopen* y cerrarlos con *fclose*

```
FILE *fopen(const char *path, const char *mode);
int fclose(FILE *fp);
```

- Las funciones para entrada y salida a fichero son

```
int fprintf(FILE *stream, const char *format, ...);
int fscanf(FILE *stream, const char *format, ...);
char *gets(char *s);
int fputs(const char *s, FILE *stream);
int fgetc(FILE *stream);
int fputc(int c, FILE *stream);
```

Biblioteca C

Entrada/salida con formato a memoria

- Existen también funciones que nos permiten simplemente hacer la entrada/salida sobre variables en memoria.
- Estas funciones operan exactamente igual que las que hemos visto, pero requieren un argumento adicional
  - Una variable tipo puntero a carácter, donde se va a colocar la salida formateada adecuadamente (o desde donde se pretende leer la entrada)

```
int sprintf(char *str, const char *format, ...);

int sscanf(const char *str, const char *format, ...);
```

Biblioteca C

Entrada/salida sin formato

- Quizá la manera mas cómoda de realizar e/s sea la de las llamadas al sistema en UNIX; estas llamadas simplemente leen bytes y escriben bytes en un fichero
- Si queremos que tengan un formato específico, podemos formatear previamente los datos
- Para abrir un fichero

```
int open(const char *pathname, int flags, mode_t mode);
```

pathname	nombre del fichero
flags	modo de apertura O_RDONLY, O_WRONLY, O_RDWR, O_CREAT O_EXCL...
mode	permisos del fichero (solo si se crea el fichero)

- Devuelve un entero (*descriptor del fichero*) que se usa en *read* y *write*

- Para leer o escribir

```
ssize_t read(int fd, void *buf, size_t count);
ssize_t write(int fd, const void *buf, size_t count);
```

fd	descriptor de fichero (obtenido con *open*
buf	dirección de memoria para la transferencia de datos
count	número de bytes de e/s

- Devuelven en número de bytes que se han transferido
- para cerrar el fichero

```
int close(int fd);
```

- Otras funciones relacionadas (tras abrir fichero con fopen())

```
size_t fread (void *ptr, size_t tam, size_t n, FILE *stream)
size_t fwrite(void *ptr, size_t tam, size_t n, FILE *stream)
int fseek(FILE *stream, long desplto, int origen);
long ftell(FILE *stream);
```

Biblioteca C

Ejercicios

- Realizar en C un programa que reciba como parámetro el nombre de un fichero y genere tres ficheros
  - Uno que contiene los mismos caracteres pero en orden inverso
  - Otro que contiene las mismas palabras pero en orden inverso
  - Otro que contiene las mismas líneas pero en orden inverso
- Se supone que la separación entre líneas esta dada por una ocurrencia del caracter fin de línea ('\n'), y la separación entre palabras por una o más ocurrencias del caracter espacio (' ') o del caracter *tab* ('\t')
- Puede usarse `strtok`

Herramientas

Valgrind

Depuración de memoria con Valgrind

- Valgrind es una herramienta que permite detectar fallos en la gestión de memoria
  - Uso de memoria sin asignar
  - Accesos incorrectos a memoria
  - Memoria asignada, no liberada y perdida (*memory leaks*)
- Para usar el depurador
  a. Compilamos con `-g -O0`

     ```
     $ gcc -g -O0 ejercicio1.c
     ```

  b. Invocamos valgrind pasándole el ejecutable (y sus parámetros) como parámetro

     ```
     $ valgrind --leak-check=full --show-reachable=yes  a.out
     ```

Ejemplo de depuración de memoria con Valgrind (I)

```
#include <stdio.h>
#include <malloc.h>
main() {
  int i;
  int *v;
  v=(int*) malloc (10*sizeof(int));

  for (i=0;i<=10;i++)              // aquí hay un acceso no valido!
      v[i]=i*2;

  for(i=0;i<10;i++) printf("%d",v[i]);

  // free(v);                      // no se libera memoria!!!
}
```

Ejemplo de depuración de memoria con Valgrind (II)

```
$ valgrind --leak-check=full --show-reachable=yes  a.out
==27888== Memcheck, a memory error detector
==27888== Copyright (C) 2002-2010, and GNU GPL'd, by Julian Seward et al.
==27888== Using Valgrind-3.6.1 and LibVEX; rerun with -h for copyright info
==27888== Command: ./a.out
==27888==

==27888== Invalid write of size 4
==27888==    at 0x8048448: main (p.c:11)
==27888==  Address 0x41b8050 is 0 bytes after a block of size 40 alloc'd
==27888==    at 0x4025018: malloc (in /usr/lib/valgrind/vgpreload_memcheck-x86-linux.so)
==27888==    by 0x8048428: main (p.c:8)

==27888==
0 2 4 6 8 10 12 14 16 18 ==27888==
==27888== HEAP SUMMARY:
==27888==     in use at exit: 40 bytes in 1 blocks
==27888==   total heap usage: 1 allocs, 0 frees, 40 bytes allocated
==27888==

==27888== 40 bytes in 1 blocks are definitely lost in loss record 1 of 1
==27888==    at 0x4025018: malloc (in /usr/lib/valgrind/vgpreload_memcheck-x86-linux.so)
==27888==    by 0x8048428: main (p.c:8)

==27888==
==27888== LEAK SUMMARY:
==27888==    definitely lost: 40 bytes in 1 blocks
==27888==    indirectly lost: 0 bytes in 0 blocks
==27888==      possibly lost: 0 bytes in 0 blocks
==27888==    still reachable: 0 bytes in 0 blocks
==27888==         suppressed: 0 bytes in 0 blocks
==27888==
==27888== For counts of detected and suppressed errors, rerun with: -v
==27888== ERROR SUMMARY: 2 errors from 2 contexts (suppressed: 11 from 6)
```

Ejemplo de depuración de memoria con Valgrind (III)

Tras corregir errores

```
$ valgrind --leak-check=full --show-reachable=yes  ./a.out
==12056== Memcheck, a memory error detector
==12056== Copyright (C) 2002-2010, and GNU GPL'd, by Julian Seward et al.
==12056== Using Valgrind-3.6.1 and LibVEX; rerun with -h for copyright info
==12056== Command: ./a.out
==12056==
0 2 4 6 8 10 12 14 16 18 ==12056==
==12056== HEAP SUMMARY:
==12056==     in use at exit: 0 bytes in 0 blocks
==12056==   total heap usage: 1 allocs, 1 frees, 40 bytes allocated
==12056==
==12056== All heap blocks were freed -- no leaks are possible
==12056==
==12056== For counts of detected and suppressed errors, rerun with: -v
==12056== ERROR SUMMARY: 0 errors from 0 contexts (suppressed: 11 from 6)
```