Leccion 4
Lenguaje de Programacion C

Resumen de la leccion 4

- Variables locales
- Variables globales
- mas sobre cadenas de I/O
- mas sobre I/O en archivos
- librerias de funciones
- librerias de funciones para manipulacion de cadenas
- librerias de funciones para matematicas
- librerias de funciones para la gestion de la memoria
- gestion de la memoria (dynamic memory allocation)
- arrays de 2D (2 dimensiones)

Variables locales

- Las variables declaradas dentro de una funcion son *variables internas.*
- Las variables internas declaradas sin `static` son de **clase**:

  1) locales (accesibilidad)
  2) automaticas (longevidad)

```
e.g. int j=1;
```

- Las variables internas declaradas con `static` son:

  1) locales
  2) creadas e inicializadas en *compile time*
  3) Muy util para recursiones

```
e.g. static int j=1;
```

Ejemplo con variables locales y estaticas

```
void countdown()
{
   static int ii=10;

   if (ii == 0)
        return;

   printf("%d ",ii--);
   countdown();

}

main()
{
   countdown();
}
```

```
%a.out
10 9 8 7 6 5 4 3 2 1
```

Variables externas o globales

- Las variables declaradas fuera funcion son externas o *globales*
  - Estas variables tienen *external linkage (enlace externo)*
  - Cuando usamos la misma variable global en dos archivos utilizar el modificador `extern`

```
/* file1.c */
int mycount=0;
main()
{
   int i;
   for(i=0;i<10;i++)
      printf("%d ",countinc());
}
```

```
 /* file2.c */
extern int mycount;

int countinc()
{
  return(++mycount);
}
```

```
%a.out
1 2 3 4 5 6 7 8 9 10
```

Variables globales estaticas

- En general, las variables globales son malas, especialmente en programas grandes.
- La variable global estatica es un compromiso entre las variables globales y locales.

- La variable global estatica tiene *file scope* o *internal linkage*

```
/* file1.c */
static int mycount=0;
main()
{
   int i;
   for(i=0;i<10;i++)
      printf("%d ",countinc());
}
```

```
 /* file2.c */
extern int mycount;

int countinc()
{
  return(++mycount);
}
```

Al compilar file1.c file2.c se producen errores:

```
variable mycount in file2.o
ld: fatal: Symbol referencing errors. No output written to a.out
```

Clases de almacenamiento

auto
register
static
extern

```
#include <stdio.h>

nuevovalor();
main()

{int i;
for (i=1;i<10;i++) nuevovalor();
}
nuevovalor()
{static int snum=25;
int anum=25;
printf("snum= %3d      anum= %3d \n",++snum,++anum);
}
```

E/S de cadenas (String I/O)

- `puts` escribe una cadena en la salida estandar
- `gets` lee una cadena desde la entrada estandar, el caracter "newline" (retorno) es interpretado como ‘\0’
- `fgets` y `fputs` funciona con cualquier secuencia de informacion (e.g. archivos)
- Otras ordenes para el tratamiento de cadenas:

```
char name = “Jane Smith”;
char buf[1000];
fprintf(fp,“%s”,name);
fscanf(fp,”%s”,&buf);
printf(“%s”,name);
sprintf(buf,”%s”,name);
```

E/S en archivos (FILE I/O)

- Las cuatro funciones basicas para E/S ascii en archivos:

  `fprintf`
  `fscanf`
  `fopen`
  `fclose`

- No olvidarse de fclose cada vez que se use fopen

E/S archivos binarios

- Las funciones basicas para E/S en archivos binarios son: fread,fwrite,fopen,fclose;

```
size_t fread(void *ptr, size_t s,
        size_t n, FILE *stream)
size_t fwrite(void *ptr, size_t s,
        size_t n, FILE *stream)
```

```
Ejemplo:

int size=10;
FILE *stream;
char *x;
if ((stream = fopen(name,”rb”)) == NULL
{
fprintf(stderr,”cannot open %s\n",name);

  exit(1);
}
fread(x,size,1,stream);
fclose(stream);
```

Librerias de funciones

- Las librerias de funciones son conjuntos de funciones utiles que han sido predefinidas.
- El C tiene varias librerias de funciones estandar
- Para usarlas se escribe, #include "nombre_libreria".h y el enlace se realiza durante la compilacion.
- Ejemplo de archivos *.h que son empleados a menudo:

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

- La no inclusion de estos archivos puede producir problemas. A menos que se incluya #include <math.h> en el siguiente programa, el resultado sera 0.000

```
main()
{
 printf("2 cubed is %f\n",pow(2,3));
}
```

Librerias estandar

- Matematicas: #include <math.h>
- Manipulacion_de_cadenas:#include<string.h>
- Entrada/Salida:#include <stdio.h>
- Dynamic Memory Allocation #include <stdlib.h>

Libreria de funciones de cadenas

`char *strcat(s,cs)`	Concatenates a copy of cs to end of s; returns s.
`char *strncat(s,cs,n)`	Concatenates a copy of at most n characters of cs to end of s; returns s;
`char *strcpy(s,cs)`	Copies cs to s including \0. returns s.
`char *strncpy(s,cs,n)`	Copies at most n characters of cs to s; returns s; pads with \0 is cs has less than n characters
`char *strtok(s,cs)`	Finds tokens in s delimited by characters in cs.
`size_t strlen(cs)`	Returns length of cs (excluding \0)
`char *strcmp(cs1,cs2)`	Compares cs1 and cs2; returns negative, zero, or positive fo cs1 <,==, or > cs2 respectively
`char *strncmp(cs1,cs2,n)`	Compares first n characters of cs1 and cs2; returns as in strcmp.
`char *strchr(cs,c)`	Returns pointer to first occurrence of c in cs
`char *strrchr(cs,c)`	Returns pointer to last occurrence of c in cs
`char *strpbrk(cs1,cs2)`	Returns pointer to first char in cs1 and cs2
`char *strstr(cs1,cs2)`	Returnss pointer to first occurrence of cs2 in cs1 The 4 above functions return NULL if search fails
`size_t strspn(cs1,cs2)`	Returns length of prefix of cs1 consisting of characters from cs2
`size_t strcspn(cs1,cs2)`	Returns length of prefix of cs1 consisting of characters not in cs2

Libreria Matematica

- Ejemplos de funciones matematicas incluidos en la libreria matematica: cos,sin,sqrt,log,pow

Problemas con los arrays estaticos

- Supongamos que tengo que programar lo siguiente:
  - Leer un conjunto de precios de un archivo
  - El primer numero es el numero de precios que van a ser leidos
  - No hay limite en el numero de precios que hay en el archivo
- ¿Como dimensiono el array de variables?
- La respuesta *dynamic memory allocation*

- `malloc` es la funcion para reservar memoria

```
extern void *malloc(size_t);
```

- free es la funcion para liberar memoria una vez finalizado

```
extern void free(void *);
```

```
/* Ejemplo sencillo de dynamic memory allocation */
main () {
  char *filename = "price.dat";
  FILE *fp;
  int numprice;
  float *pricedata;
  int j;


  /* Apertura del archivo*/
  if ((fp = fopen(filename,"r")) == NULL) {
        fprintf(stderr,"cannot open %s\n",filename);
        exit(-1);
  }

  /* leer el numero de precios que van a ser leidos */
  fscanf(fp,"%d",&numprice);

  /* reservar sitio para los precios */
  pricedata = (float *) malloc(sizeof(float)*numprice);

  /* lectura de los precios desde el archivo */
  for (j=0;j<numprice;j++)
     fscanf(fp,"%f",pricedata++);

  /* cierre del archivo */
  fclose(fp);
}
```

Precaucion:

Perdida de memoria (Memory Leak):
una funcion usa malloc sin free ...
puede que no haya mas memoria !!

Punteros colgados (Dangling Pointers):
un puntero apunta a memoria sin reserva.

Varios

- void * es un puntero generico (los punteros a cualquier tipo de dato son del mismo tamano (son direcciones))
- malloc devuelve (void *) y debe definirse el tipo para determinar la clase de "puntero a"
- OJO: cuando hemos utilizado pricedata en lugar de &pricedata en la orden scanf (scanf necesita un puntero y pricedata es un puntero)
- Observar que el incrementador usado en pricedata, es un metodo eficiente de recorrer el array de precios, pero tened cuidado con los punteros "colgados" que se pueden ir fuera de la longitud del array

Encontrar los errores de punteros

- `int *pa;` `/* pa es un puntero a entero */`
- `*pa = 1;` `/* A pa se le da el valor 1 */`
- `int *pb = pa;` `/* se declara pb como puntero a entero y se le da la misma posicion de memoria que tiene pa */`
- Considerar :

```
int *pa, *pb;
*pa = 1;          // Error!
```

- Considerar:

```
int *pa;
int foo;
pa = &foo;
printf(“%d”,*pa); // Bien
pa = 1;
printf(“%d”,*pa); // Error !!
```

Repaso de la doble indireccion

- Considerar lo siguiente:

```
int foo;
int *pa ;
int **ppa;
```

- Que tipo es `&foo`? Es un puntero a entero;

```
int *pa = &foo;
```

- Que tipo es &pa? Es un puntero a puntero;

```
int **ppa = &pa;
```

- Despues de: `pa = &foo`; Lo siguiente es equivalente:

```
*ppa = pa;    *ppa = &foo;
```

- Es correcto esto?

```
int **ppa = &&foo;
```

Arrays de 2D

- Este es un array de 10 x 20 elementos

  `int arr[10][20];`
- arr es ahora lo mismo que &(arr[0][0])
- Hay 10 filas y 20 columnas.
- Los datos se almacenan en un formato de filas

  `arr[2][5]` es lo mismo que

  `arr[0][2*20 + 5]`
-

Nombre	Tipo	Igual que
arr	puntero a puntero	&(arr[0])
arr[0]	puntero a entero	&(arr[0][0])
arr[2]	puntero a entero	&(arr[2][0])
arr[2][5]	entero	arr[0][2*20+5]

- Obsevar como se puede usar el puntero a puntero:

```
int **ppa;
ppa = arr;   /* ppa apunta a arr[0]; */
ppa = arr[2] /* ppa apunta a arr[2] */
*((*ppa) + 3) = 7;
/* en arr[2][3] se almacena un 7 */
  **(ppa +3) = 7;
/* en arr[5][0] se almacena un 7 */
```

Arrays y punteros (1)

El nombre de un array sin más, es la dirección donde se encuentra el primer elemento, pero es una constante, no es un puntero ya que tendría que poder almacenar valores variables.

argv ≡ &argv[0] ,argv no es un puntero.

Si al nombre del array le aplicamos operadores aritméticos, éstos actúan automáticamente,

argv[5] ≡ ∗(argv+5)

y además más rápido:

si, ptr=argv es más rápido *(ptr++) que argv[i++]

Generalizando, si se declara *int a[10]* y *int *pa* y *pa=&a[0],* se cumple que *pa+i* es la dirección de *a[i]* y **(pa+i)* es el contenido de *a[i].*

Arrays y punteros (2)

Por definición *a* es igual *&a[0]* y por tanto en C al evaluar *a[i]* lo convierte en **(a+i)*.

Pero un puntero como *pa* es diferente a *a* ya que el primero es una variable y el otro es constante, con lo que operaciones como *pa=a* y *pa++* son correctas y *a=pa* ó *a++* son ilegales.

Las definiciones *char s[ ]* y *char *s* son equivalentes.

La fuerza real de la aritmética con punteros radica en su uso con arrays, ya que se gestionarán de forma automática, independientemente del tipo de variables que almacene el array.

En C la integración de punteros, arrays y aritmética de direcciones es una de sus principales virtudes.

Es ilegal cualquier operación aritmética con punteros que no sea sumar o restar un entero y restar o comparar dos punteros.

Arrays y punteros (3)

Un paso adicional sería convertir un array de arrays en un array de punteros:

*int **argv* ,se declara argv como puntero a puntero o sea, argv apunta al elemento cero de un array, que es un puntero, y

argv[0][0] " ***argv* y

&argv[0][0] " *argv[0]* " **argv* luego,

**(*argv+1)*" *argv[0][1]* que es distinto de

***(argv+1)*" *argv[1][0]*

Para el programador principiante, los arrays se manejan mejor como arrays y no como arrays de punteros. La ventaja de usar estos últimos reside en su capacidad de acelerar muchos procesos.

Arrays y punteros (4)

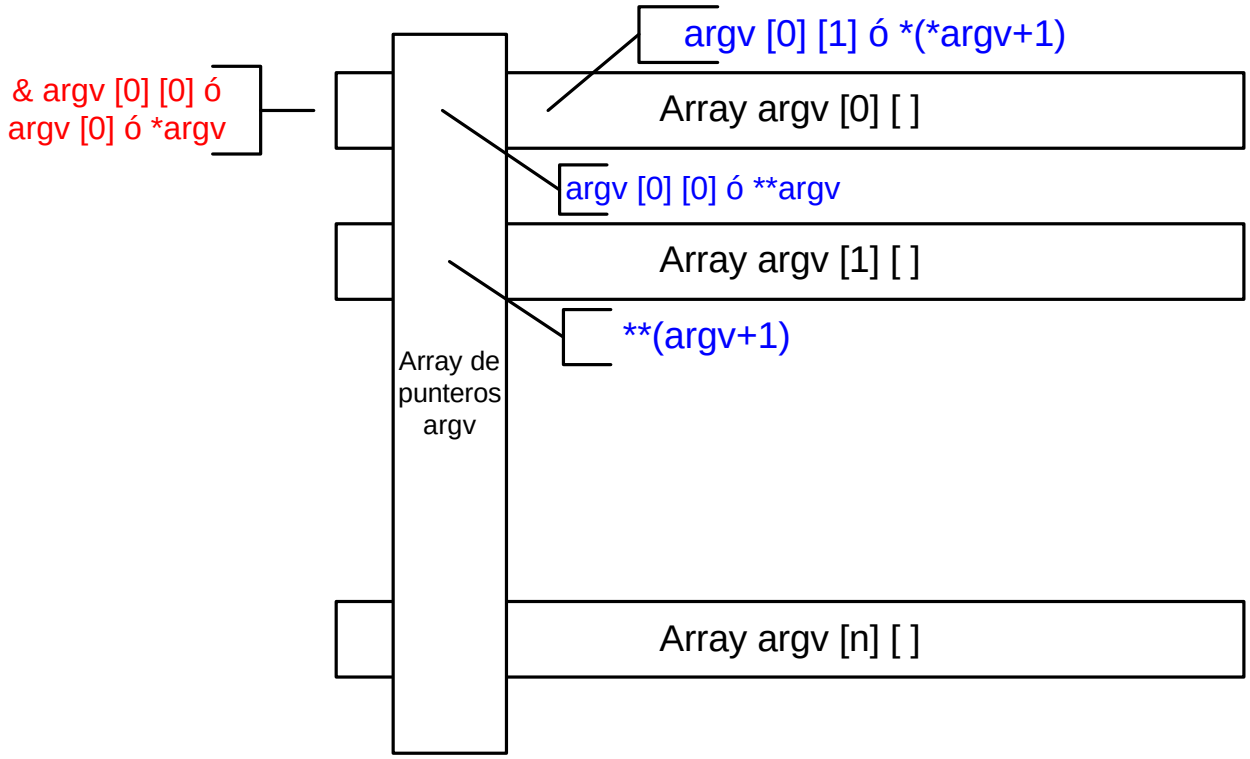
Como conclusiones:

char s[]; s++; es incorrecto y char *s; s++; es válido.

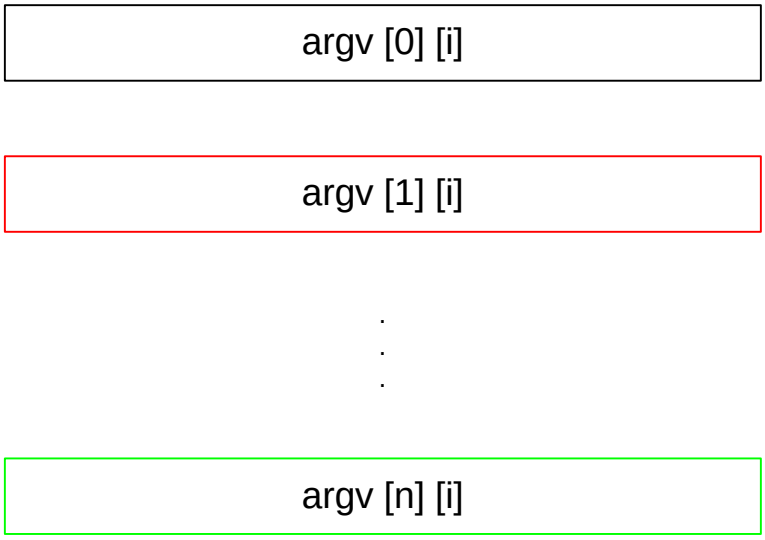
Para pasar la dirección de una fila, se pasa la dirección del elemento cero de la misma: &argv[fila][0] o abreviadamente: argv[fila].

Es preferible por el ahorro de espacio en memoria, definir un array bidimensional usando un array de punteros: char *nombres[ ]; que no un array convencional: char nombres[ ][ ];.,ya que en el primer caso la longitud de las filas se ajusta al número de elementos necesarios en esa fila (“matriz irregular”) y en el segundo todas las filas tienen el mismo número de columnas, sean necesarias o no (“matriz regular”).

Direcciones Contenidos del array argv [ ] [ ]



Representación de un array 2D (bidemensional) n x i.



Es equivalente a: