CC3301
Programación de Software de Sistemas
Profesor: Luis Mateu

- Aritmética de punteros
- Declaración de arreglos con inicialización
- Errores comunes con punteros y arreglos
- Strings
- Strings constantes
- Funciones para manipular strings
- Errores comunes con strings

Resumen de la clase pasada: variables y punteros

- Una variable reside en la memoria
  - Almacena valores de un tipo específico
  - Se puede declarar, asignar y evaluar
  - También ***se puede obtener su dirección***
  - Y su tamaño en bytes
- Un puntero es una variable que almacena direcciones de variables
  - Sirven para implementar los strings, las estructuras de datos y mucho más
  - El operador de contenido permite acceder a la variable a la cual apunta un puntero
- Un arreglo es un conjunto enumerado de variables del mismo tipo
  - El operador de subindicación permite seleccionar una de esas variables por medio de un índice
  - Hay una equivalencia entre subindicación y aritmética de punteros

Aritmética de punteros

- double z[100];
- Todos los elementos son del mismo tipo
- Ocupan direcciones contiguas en la memoria
- Para obtener la dirección del i-esimo elemento de *z*:

```
int i = 23;
double *p = &z[i];
```

- El identificador *z* representa la dirección del primer elemento:

p = z; ⇔ p = &z[0];

- En la expresión *dir [índice]*
  - *índice* puede ser cualquier expresión de tipo entero
  - *dir* puede ser cualquier expresión de tipo puntero (T *)
  - ¡También puede ser un puntero! Ejemplo: *p[i]*
  - Accede a la variable de dirección *dir* + *índice* * sizeof(T)
  - ¡El índice puede ser negativo!
  - Si la dirección cae fuera de un segmento del programa ⇒ Segmentation fault
- Equivalencias: & *dir [índice]* ⇔ *dir + índice*

  & *dir [- índice]* ⇔ *dir - índice*

Aritmética de punteros

Maneras rebuscadas de inicializar un arreglo en 0, pero correctas

```
// Versión tradicional
double z[10];
for (int i= 0; i<10; i++) {
        z[i]= 0;
}
```

```
// Versión rebuscada
double z[10];
double *p= z+5; // &z[5]
for (int i= -5; i<5; i++) {
    p[i]= 0;
}
```

¡No haga esto!

Notas:

i++ ⇔ i += 1

i += 1 ⇔ i = i+1

*p++ =0; ⇔ *p= 0; p++; *postincremento*

*++p =0; ⇔ ++p; *p= 0; *preincremento*

```
double z[10];
double *p = z, *top= z + 10;
while ( p < top ) {
    *p++ = 0;
}
```

Esta versión *era* ligeramente más eficiente que la tradicional: se usa mucho

¡Las direcciones se pueden comparar!

Cuidado con los arreglos

La declaración: char s[10];

- Atribuye espacio en memoria para un arreglo de 10 caracteres
- *s* representa la dirección del primer elemento del arreglo
- Pero *s* no es una variable, representa un valor
- No se puede cambiar la dirección con una asignación:

  s = …; // ¡incorrecto!

- Por la misma razón que una constante no se puede cambiar con una asignación:

  1000 = …; // ¡incorrecto!

- Pero un puntero sí se puede asignar

  char *p = s; // p almacena *&s[0]*

  p = p + 5; // correcto porque *p* sí es una variable

- Pero la declaración *char *p;* ***solo atribuye espacio para el puntero, nunca para lo apuntado***

  char *p; p[4]= 0; *// ¡incorrecto! ¡No lo haga!*

Más sobre aritmética de punteros

- Si *p* y *q* son punteros o arreglos, las siguientes expresiones son inválidas: *p*5* *p+2.5* *p/10* *p+q*
- Solo tiene sentido *p* + o – una expresión entera
- Si *p* y *q* son punteros del mismo tipo:

  p – q es correcto y es de tipo entero
- El valor de *p – q* satisface:

  p – q = i *sí y solo sí* *p = q + i*

Más sobre arreglos y punteros

- Declaración con inicialización:

```
int a[ ] = { 2, 3, 5, 7, 11, 13 }; // Arreglo de 6 elementos
int b[100] = { 2, 3, 5 } ;      // solo inicializa los 3 primeros
                                // de un arreglo de 100 elementos
b = { 4, 8, 12 }; // Incorrecto, no es una declaración
int c[ ];             // Incorrecto, no se puede inferir el tamaño
```

- Una función no puede recibir como parámetro un arreglo, pero sí puede recibir un puntero y operarlo como si fuese un arreglo
- Si se declara una función con este encabezado:

```
void fun(int arreglo[ ]);  //  ¡Es legal!
```

- Pero el compilador silenciosamente cambia el encabezado por:

```
void fun(int *arreglo);
```

Strings

- *Un string es un arreglo de caracteres que termina con un byte que almacena el valor 0*: no '0'
- Cuidado: 48 = '0' ≠ 0
- Ejemplo: char str[ ] = {'H', 'o', 'l', 'a', 0};
- Se referencian por medio de la dirección de su primer caracter
- Ejemplo: printf("%s\n", **str**);
- Se puede asignar a un puntero: char *r = str;
- Ejemplo: contar las letras mayúsculas:

```
char *r = str;
int cnt = 0;
while (*r != 0) {
      if ('A' <= *r && *r <= 'Z')
            cnt++;
      r++;
}
printf("Mayúsculas: %d\n", cnt);
```

Strings constantes

- Todo lo que se escribe entre " … "
- Ejemplo: char *str2 = "Hola";
- ¡No se pueden modificar! *str2 = 'h'; // Seg. Fault
- Se almacenan en un área de memoria de solo lectura
- Sintaxis especial para declarar strings mutables:

```
char str3[ ] = "Hola";  // No estaba en el C original
*str3 =  'h';  // Correcto
printf("%d\n", str3);  // Muestra hola, h minúscula

str3 [ ] = "Hello";   // Error sintáctico
```

Funciones para manipular strings

- int strlen(char *s): calcula el largo de un strings, sin contar el 0 que lo termina

  strlen("Hola") es 4

  no entrega el tamaño de memoria atribuido

- char *strcpy(char *d, char *s): copia el string s en el string d

  ```
  char d[20];
  strcpy(d, "Hola");
  ```

  El destino d debe ser la dirección de un área de tamaño suficiente (largo del string + 1)

  ```
  char *p;
  strcpy(p, "Hola");     // Incorrecto, ¿seg. Fault?
  ```

  porque *p* no ha sido inicializado con ninguna dirección válida

  ```
  char s[ ] = "Hola";     ⇔     char s[strlen("Hola")+1];
                                strcpy(s, "Hola");
  ```

Comparación de strings

- int strcmp(char *s, char *r): compara los strings *s* y *r* retornando 0 si son iguales, < 0 si *s* es lexicográficamente menor que *r* y > 0 si es mayor

```
char s[20]= "juan";
strcmp(s, "pedro")      es < 0
strcmp(s, "diego")      es > 0
strcmp(s, "juan")       es 0
strcmp(s, "Juan")       es > 0
```

- *Cuidado con los operadores relacionales == != < > <= >= porque comparan direcciones, no contenidos*

```
s == "juan"             es ≠ 0
```

Implementación de strlen y strcpy

```
int mistrlen(char *s) {
  char *r= s;
  while (*r++)
    ;
  return r-s-1;
}
```

```
char *mistrcpy(char *d, char *s) {
  char *t= d;
  while (*t++ = *s++)
    ;
  return d;
}
```

No promuevo este estilo de código, pero deben aprender a entender este estilo porque hay mucho código en C escrito así