Ayudantía 2 - Estructura de Datos y Algoritmos ELO320

Funciones, estructuras, arreglos y punteros en C

Felipe Vera A.

24 de marzo de 2014

1. Directivas de precompilador

Corresponden a instrucciones que se ejecutan antes de compilar, haciendo que ciertas rutinas se vuelvan más cómodas de programar.

- `#include nombre_arc`: Sirve para incluir el contenido de un archivo fuente. Si están en la misma carpeta, el nombre de archivo se encierra entre comillas y si es una librería (como `stdio.h`) se encierran entre $<$ y $>$.
- `#define`: Sirven para:
  - **Definir constantes**: Es una buena costumbre definir constantes de esa manera, de modo que se pueda cambiar el código sólo cambiando sólo este `define`. Un ejemplo:

    ```
    #define MI_CONSTANTE    36
    ```

  - **Escribir macros**: Las macros son similares a funciones sencillas. Son útiles para evitar escribir código repetitivo.

    ```
    #define CAMBIAR_SIGNO(x)    x * -1
    ```

2. Arreglos

Son un método para ordenar un listado de datos, a los cuales se puede acceder con un índice. Lo que permite esta facilidad es que los elementos de un arreglo estén contiguos en memoria. Se declaran asignando una cantidad fija de memoria, la cual posteriormente no se puede cambiar.

```
int lista[5];
lista[0] = 22;
[...]
lista[4] = -15;
```

Como se puede ver arriba, se declaró un arreglo de 5 enteros, pudiendo acceder a ellos usando índices del 0 al 4. El arreglo `lista` también se puede declarar de la siguiente manera:

```
int lista[] = {22, 3, 25, -4, -15};
```

Si se proporcionan los datos al momento de declarar la variable, el compilador automáticamente reconoce el largo que deberá tener el arreglo para poder alojar a todos esos elementos.

2.1. Cadenas de caracteres

Los arreglos de tipo `char` son una forma conveniente de trabajar con texto en C. En este caso cada número representa una letra[1].

Al momento de crear una cadena de caracteres, el compilador siempre asigna como elemento final un caracter cero (`\0`) para que las funciones que trabajan con texto encuentren el final de la cadena. Por ejemplo:

```
char cadena[] = "Perro";
```

Esto reserva la cantidad de letras que componen la palabra `Perro` más el caracter cero. En total: 6 elementos.

2.2. Algunas precauciones

No hay forma de saber la cantidad de elementos -o memoria asociada- que tiene un arreglo, por lo que a través del código se deberá recordar esto. Si se accede a un índice inexistente, ocurre un error llamado *Falla de segmentación* o *Segmentation fault* y el programa se cierra.

3. Punteros

Un puntero almacena la dirección de una variable en la memoria, lo que permite manipularla de todas las formas imaginables.

3.1. Declaración

Se pueden declarar como cualquier tipo de datos, colocándoles un asterisco (`*`) delante.

```
char *puntero1;
int *puntero2;
```

3.2. Uso de punteros

Si se piensa a un puntero como a una casilla en donde se puede alojar un dato, si se antepone el operador asterisco `*` se puede acceder o modificar el contenido de la memoria a la que apunta el puntero.

Asimismo, si a una variable se le antepone el operador *ampersand* `&`, se obtiene la dirección de memoria de esa variable.

```
int variable;
int *puntero;

variable = 53;
puntero = &variable;       //Ahora "puntero" tiene la dirección de "variable"
*puntero = 42;             //Se modifica el contenido de la memoria asociada a "puntero",
                           //la cual coincide con "variable"
printf("%d", variable);    //Debería mostrar 42.
```

Otros grandes usos para los punteros son los siguientes:

- Uso en funciones para modificar el valor de variables (Paso por referencia).
- Arreglos (los arreglos son en realidad punteros).

[1] Busque *Código ASCII en internet*

- Asignar memoria dinámicamente (se verá más adelante).
- En programación de microcontroladores, se usan punteros para modificar registros que controlan funciones especiales (como arrojar 5V ó 0V desde un pin)

Cuidado Al declarar punteros, no se les asigna memoria automáticamente. El código a continuación está mal.

```
int *puntero_a_nada;
*puntero_a_nada = -32;    //Esto está malo. Arrojará un error y cerrará el programa.
```

Sin embargo, en la próxima ayudantía se verá cómo reservar memoria para un puntero.

4. Estructuras

Sirven para agrupar patrones de datos de distintos tipos. Un ejemplo sencillo puede ser los datos de un enemigo en un videojuego.

```
struct enemigo
{
    int puntos_vida;
    int dano;
    float critico;
    int nivel;
};
```

Con esto se crea un nuevo "tipo de dato" llamado `struct enemigo`. Se pueden declarar de forma similar a los arreglos, asignando valores a sus elementos ocupando llaves y separando en comas.

```
struct enemigo caballero = {1200, 42, 0.3, 10};
```

Esto creará un `caballero` con 1.200 puntos de vida, 42 de daño, 0,3 probabilidad de crítico y nivel 10. Se pueden leer o modificar separándolas con un punto (`.`). Supongamos que queremos subirlo de nivel.

```
caballero.nivel = 11;
caballero.nivel++;        //Una alternativa.
```

5. Funciones

Las funciones son un gran método para reutilizar código y son muy dinámicas. Este dinamismo se logra gracias a los parámetros que se les entregan.

5.1. Cómo escribirlas

Se debe tener en cuenta el tipo de datos que se quiere devolver (`int`, `char`, `float`...). Si no se quiere devolver ningún dato, se ocupa `void`. A continuación se incluyen los argumentos de entrada a la función, los cuales pueden ser de dos tipos:

- **Por valor**: Cuando se solicita un tipo de datos que no es puntero ni arreglo, se le proporciona un argumento por valor a la función. Se copia los contenidos de dicha variable dentro de la función, de modo que no se vea alterada fuera de ella.

- **Por referencia**: Cuando se solicita un puntero o arreglo. Generalmente se ocupan para que la función pueda modificar los contenidos de una variable que se ocupa fuera de la función, como en la función `scanf`.[2]

Se debe tener cuidado en qué lugar del código se ubican las funciones. Si estas son escritas luego del lugar en el que son usadas, el compilador no la reconoce y arroja un error. Eso se puede solucionar de dos maneras:

1. Escribir la función antes de que sea llamada.
2. Escribir un *prototipo* de la función antes de que sea llamada, de modo que el compilador sepa que la función existe y cuáles son sus argumentos.

Un ejemplo es el siguiente:

```
int funcion(int num, char *cad);

int
main(int argc, char **argc)
{
    [...]
    funcion(1, "Hola");
}

int
funcion(int num, char *cad)
{
    printf("%s %d\n", cad, num);
    return num++;
}
```

Ejercicios

1. MACROS: Defina una macro que le reste 1 a un número si es impar.
2. ARREGLOS: Cree un arreglo de 50 enteros dentro de la función `main`, asígnele a cada uno de sus elementos un número aleatorio (busque sobre la función `rand` en la referencia). Luego almacene e imprima con `printf` cada elemento tras procesarlo con la macro escrita en el punto 1.
3. CONSTANTES: Modifique el código anterior para que el largo del arreglo pueda modificarse mediante un `#define`.
4. FUNCIONES, PUNTEROS: Cree una función `sumararreglo` con tres argumentos: un arreglo, el largo del arreglo y un entero, de modo que recorra todo el arreglo sumando este entero a todos los elementos del arreglo.
5. VALORES DE RETORNO: Modifique el código anterior para que la función `sumararreglo` devuelva la suma de todos los elementos del arreglo. Imprímala en `main`.

[2] Vea la referencia de funciones de la librería estandar de GNU para más información sobre `scanf`.

Desafío

Se sabe que la distribución normal tiene la función de densidad, donde μ es la esperanza, y σ es la desviación estandar de la distribución.

$$f_X(x) = \frac{1}{\sigma\sqrt{2\pi}} \cdot e^{\frac{-(x-\mu)^2}{2\sigma^2}}$$

Para convertir de una distribución uniforme (como la generada por `rand`) a una distribución normal con μ y σ dados, donde y es el número generado con la distribución normal, x es el número generado con la distribución uniforme, M es el máximo número aleatorio que es posible generar y *erf* es la función error.

$$y = \sqrt{2}\sigma \cdot \text{erf}^{-1}\left(2\frac{x}{M} - 1\right) + \mu \quad \approx \quad \sqrt{2}\sigma \cdot \arcsin\left(2\frac{x}{M} - 1\right) + \mu$$

Cree siguiente función, de modo que almacene una cantidad de números generados aleatoriamente dada por `n_elementos` en el arreglo `arreglo_in`, con esperanza μ y desviación estándar σ.

```
void
dist_norm (float *arreglo_in, int n_elementos, float mu, float sigma)
{
    [...]
}
```

Recuerde Incluir la librería `math.h` y añadir como argumento a GCC al momento de compilar la opción `-lm` para poder usar funciones matemáticas como las trigonométricas inversas.