Estructura de Datos y Algoritmos
Departamento de electrónica
Universidad Técnico Federico Santa María

DEPARTAMENTO DE ELECTRONICA

Ayudantía 3: Librerías, memoria y otros alcances sobre C

Objetivo Principal: Conocer otros temas útiles y complementarios sobre C.
Autor: Kevin Pizarro Aguirre

---

1. Librerías adicionales

Hasta el momento se han utilizado funciones de la librería **stdio.h**, para imprimir y leer datos principalmente. Pero también existe múltiples librerías que son útiles, se verá rápidamente **unistd.h**, **time.h**, **string.h**, **math.h** y **stdlib.h**. Para mayor información consultar la documentación respectiva.

Unistd.h Es aquella que provee funciones, tipos de datos y constantes POSIX (Portable Operating System Interface), para una interacción con el sistema operativo. Puede ser interesante para crear, eliminar o manejar procesos en el sistema operativo (materia de otros ramos). Además, nos provee la función sleep() para poder hacer una pausa o espera activa dentro de nuestro programa.

- sleep(unsigned int): Función que detiene la ejecución del programa por los segundos que se les ingrese como parámetro.

Time.h Esta librería ayuda a manejar datos y procesos que tienen que ver con el tiempo, por ejemplo el manejo de fechas e incluso mediciones de tiempo de ejecución.

- clock_t clock(void): Entrega la marca de tiempo de del proceso actual, es decir, el programa en ejecución.

String.h Permite manipulación sobre strings de manera amistosa. Algunas de sus funciones.

- strcat(char * dest , const char * source): Concatena un string source a un string dest. En Python dest = dest+source.
- strchr(const char *str, int c): Retorna un puntero a la primera ocurrencia del caracter c dentro del string.
- strcmp(const char *str1, const char *str2): Compara el contenido de un string 1 con el de un string 2.
- strcpy(char * dest, const char * source): Copia el contenido de un string source en un string dest.
- strlen(const char *): Calcula el largo del string.

Math.h Como su nombre lo indica, contiene definiciones de funciones matemáticas y constantes útiles dentro del campo (valor de pi, e o infinito). Son muchas las funciones que se pueden encontrar, además de ser ya conocidas, por lo cual se mencionan unas cuantas: sin(), cos(), atan(), tan(), ceil(), floor(), exp(), log10(), etc.

Stdlib.h Sin dudas la más útil en conjunto a stdio.h. Su nombre proviene de *standard library*, librería estándar, por ello es que es una recopilación de funciones generales que son útiles para el programador. Dentro de las más usadas se tienen:

- abs(int number): Calcula el valor absoluto de un número entero.
- atoi(const char *str): Convierte un string a un entero.
- calloc(size_t nitems, size_t sizeofitem): Asigna la memoria solicitada y luego entrega un puntero a ella. Además inicializa sus valores a 0.
- exit(int status): Termina el programa.
- free(void *ptr): Libera la memoria solicitada previamente por calloc, mallor o realloc en el puntero designado.
- malloc(size_t size): Asigna la memoria solicitada y luego entrega un puntero a ella. A diferencia de calloc, no inicializa sus valores.
- rand(void): Retorna un número entero pseudo-aleatorio entre 0 y la constante RAND_MAX.
- realloc(void *ptr, size_t size): Intenta cambiar el tamaño de un bloque de memoria que fue previamente asignado mediante malloc o calloc.

Notar que malloc puede tener un comportamiento similar a calloc si se utiliza en conjunto a memset, inicializando así sus valores en 0. En la mayoría de las ocasiones se prefiere malloc puesto que es más rápido que calloc, pero en consecuencia puede que tenga datos basura dentro o no tenga nada.

2. Memoria

Se sabe que la memoria está dividida en pequeños bloques llamados bytes, en ellos se pueden almacenar valores de variables por ejemplo. Cada uno de los bloques tiene una dirección asociada para poder identificarlos y acceder a ellos, así como una persona tiene una dirección a la cual se le pueden dejar paquetes. También cabe destacar que se pueden distinguir dos tipos de memorias, la estática y la dinámica, según el uso y cómo se declaren las variables en el programa.

Memoria estática

Es aquella que no puede ser modificada en tiempo de ejecución del programa, puesto que se asigna a la memoria **stack** en el momento de compilar el programa. Una de las ventajas que tiene es que es más rápida y simple que la asignación de memoria dinámica, pero por contra parte es menos eficiente. Su implementación es tal cual como se ha hecho hasta el momento, por ejemplo **int i;**. Finalmente se libera su uso al finalizar el programa.

Memoria dinámica

Es aquella que puede ser asignada, liberada y modificada en tiempo de ejecución del programa, ya que se asigna a la memoria **heap** en el transcurso de ejecución del programa y cuando se requiera. Frente a la asignación en memoria estática se tiene como ventaja que es más eficiente, puede modificada en el transcurso del programa pero como desventaja tiene que es un poco más lenta, "compleja" de implementar y hay que tener cuidado con las *fugas de memoria*. Su implementación se realiza bajo los comandos calloc o malloc vistos en la sección de librería.

```
#include <stdlib.h>
void main(void)
{
    int * p;
    p = (int*)malloc(5 * sizeof(int)); // Solicita el espacio para almacenar 5 numeros enteros
    ...
}
```

Fuga de memoria Este concepto surge cuando solicitamos espacio en la memoria heap pero no la liberamos incluso cuando ya ha terminado el programa, esa sección queda "inutilizable". Esto se soluciona con la funcion free, vista en la sección de librerías, liberando el espacio solicitado anteriormente antes de que finalice la ejecución del programa.

Existen varias formas para detectar las fugas de memoria, se recomienda utilizar la herramienta Valgrind (disponible en Aragorn). Es tan simple como ejecutar el programa bajo la supervisión de Valgrind. Suponiendo que nuestro archivo ejecutable se llama "a.out" la ejecución será:

```
$ valgrind ./a.out
```

3. Otros

3.1. Función main con argumentos

La función main no es la excepción a las reglas de las funciones, también se le pueden pasar parámetros de entrada, pero la pregunta es cómo. Una definición por convención para su implementación es con **int main(int argc, char const *argv[])**, donde argc corresponde al contador de argumentos y argv al valor de los argumentos. Cuando el programa se va a ejecutar por línea de comandos siempre se le está pasando al menos 1 argumento, el nombre del archivo ejecutable, por lo cual si separamos por espacio y añadimos otra cosa estaremos añadiendo un argumento que puede ser leído y utilizado en nuestro programa.

```
$ ./a.out argumento2 argumento3 ... argumento_n
```

Esto suele ser útil cuando queremos entregar el nombre de un archivo a leer o el nombre de un archivo de salida por medio de la línea de comandos.

Conversión de tipo

Es un proceso de conversión de tipo de dato, por ejemplo de float a int. Se puede diferenciar en dos grandes familias, la conversión explícita e implícita. Para la conversión implícita, no es necesario utilizar una palabra reservada para ejecutarla, funciona entre tipos de datos compatibles en ascendencia en espacio requerido (char − > short int − > int − > unsigned int ... − > long double) simplemente se asigna como se indica en el ejemplo.

```
void main(void)
{
    int number = 1;
    char character = 'k'; //valor ASCII = 107
    int result = number + character; //result =108
}
```

Para la conversión explícita vamos en sentido opuesto generalmente, indicando que queremos realizar una conversión "forzadamente", teniendo en cuenta que en ocasiones puede existir pérdida de información. Por ejemplo, al pasar de un número decimal a un número entero se deberá truncar perdiendo los valores decimales. Su implementación es de la siguiente forma.

```
void main(void)
{
    float number1 = 1.2425;
    int number2 = (int) number1; //number2 = 1
}
```

3.2. Archivos de cabecera y archivos de código fuente

En C existen los archivos de cabecera (header files) y los archivos de código fuente (source code), esto para poder organizar y modularizar de mejor forma nuestro código y los programas que realicemos. En códigos pequeños no tiene mucho sentido, pero sí en códigos medianos o grandes proyectos, donde la lógica puede estar separada y luego se implemente toda en un archivo principal.

Por ejemplo, si se deseara programar un sistema de compra-venta con inventario para un almacén o negocio pequeño, entonces sería útil separarlo en funcionalidades o según operaciones dentro del sistema. Una función para compras, otra para ventas, otra para el inventario, otra para el manejo de los objetos u otras que se estimen convenientes. Existen buenas prácticas, metodologías y convenciones para poder diseñar la estructura de un programa, pero no es el enfoque principal del curso. Pero si cabe destacar que ante cualquier problema complejo es bueno utilizar los principios de **dividir y conquistar** y el principio **KISS** (Keep It Simple, Stupid), es decir, dividir un problema complejo en pequeños problemas y además mantener el programa lo más simple y legible posible.