AYUDANTÍA C (2DA PARTE)

Raúl Andrés Álvarez Esteban
Ricardo Esteban Schilling Broussaingaray
Ariel Martínez Silbestein
Raimundo Martínez Rioseco

IIC2333 [2020-1] - Sistemas Operativos y Redes

INTRODUCCIÓN

INTRODUCCIÓN

Los objetivos de esta ayudantía son conocer y aprender a usar:

- Header files y modularización.
- Structs o estructuras.
- Compilación con Makefile.
- Uso de `#include` y `#pragma once`.
- Conocer errores típicos.
- Uso de `errno`.

HEADER FILES (.h)

`.c` v/s `.h`

En el lenguaje de programación C, toda librería se escribe con al menos dos archivos: un `.c` y un `.h`.

El archivo `.h` corresponde a un **header file**. Este solo debe contener estructuras, enumeradores y la **declaración** de las funciones. En este se debe importar el archivo `.c` respectivo.

El archivo `.c` corresponde a un archivo de código escrito en lenguaje C, que contiene la **definición** de las funciones.

Para utilizar la librería, se deberá importar únicamente el header file, que (internamente) ya importa las funciones como tal.

HEADER FILES

Sirven para modularizar código y mantener el orden. Junto con el comando `#include`, permiten obtener acceso a funciones. Es similar a usar `import` en Python.

Algunos header files usados comúnmente son:

```
stdio.h // Manejo de I/O (printf, scanf)
stdlib.h // Manejo de memoria (*alloc, free, NULL)
string.h // Operaciones de strings
signal.h // Manejo de señales
sys/types.h // Tipos de datos, como PID
unistd.h // API de POSIX (fork, execvp)
```

También podemos escribir nuestros propios header files.

HEADER FILES

Para importar un archivo built-in:

```
#include <nombre.h>
```

Para importar un archivo propio:

```
#include "nombre.h"
```

#pragma once

#PRAGMA ONCE

El lenguaje C cuenta con un preprocesador que se encarga, entre otras cosas, de permitir que se puedan incluir header files en otros archivos. A veces, incluir múltiples veces una header file en la misma compilación puede causar errores. Esto lo podemos evitar con la siguiente directiva de preprocesador:

```
#pragma once
```

Esta directiva debe ir al principio de una header file y causa que dicha header file se incluya una única vez en toda la compilación.

STRUCTS O ESTRUCTURAS

STRUCTS O ESTRUCTURAS

En el lenguaje C no existen las clases como en Python, pero podemos lograr una aproximación usando estructuras.

Python

```python
class Persona:
    def __init__(self, nombre, edad):
        self.nombre = nombre
        self.edad = edad

p = Persona("Juanito", 22)
```

STRUCTS O ESTRUCTURAS

C

```c
struct Humano {
  int  edad;
  char nombre[30];
};

typedef struct Humano Persona;

Persona p;
p.edad = 21;
//p.nombre = "Juanito";
strcpy(p.nombre, "Juanito");
```

`typedef` lo podemos usar para darle un nombre a una estructura definida por nosotros. De esta forma podemos definir variables de nuestra estructura como si fueran un tipo de dato básico.

Por otro lado, en el ejemplo no es posible asignar directamente un `string` (entendido como un arreglo de caracteres), por lo que ocupamos el comando `strcpy`.

STRUCTS Y MEMORIA

STRUCTS Y MEMORIA

En la mayoría de los casos vamos a pedir memoria para utilizar structs, también es posible que tengamos que pedir memoria para tener structs dentro de structs. Pensemos en el sig ejemplo:

```
typedef struct mascota {
    char comida_favorita[255];
    char nombre[255];
} Mascota;

typedef struct persona {
    int edad;
    char nombre[255];
    Mascota *mascota;
} Persona;
```

Para pedir memoría para el struct `Persona` se haría lo sig.:

```
Persona *persona = malloc(sizeof(Persona));
persona->edad = 15;
strcpy(persona->nombre, "Ricardo");
persona->mascota = malloc(sizeof(Mascota));
strcpy(persona->mascota->comida_favorita, "Ají");
strcpy(persona->mascota->nombre, "Beto")
```

Sin embargo, este código es medio engorroso. ¿Como podríamos mejorarlo?

STRUCTS Y MEMORIA

```
Mascota *create_mascota(char *nombre, char *comida)
{
    Mascota *mascota = malloc(sizeof(Mascota));
    strcpy(mascota->nombre, nombre);
    strcpy(mascota->comida, comida);
    return mascota
}

Persona *create_persona(char *nombre, int edad, char *m_nombre, ch
{
    Persona *persona = malloc(sizeof(Persona));
    persona->edad = edad;
    strcpy(persona->nombre, nombre);
    persona->mascota = create_mascota(m_nombre, comida);
    return persona;
}

Persona *persona = create_persona("Raúl", 22, "Beto", "Ají");
```

¿Como lo hacemos para liberar memoria?

```c
void free_mascota(Mascota *mascota)
{
    free(mascota);
}

void delete_persona(Persona *persona)
{
    free_mascota(persona->mascota);
    free(persona);
}

delete_persona(persona);
```

Esta forma de tratar de struct es fácil de expandir y mantener, permitiendo utilizarlos sin perder legibilidad en nuestro código principal.

ENUMS O ENUMERADORES

ENUMS O ENUMERADORES

Se definen con `enum` y permiten asignar nombre a constantes enumeradas. Son útiles cuando queremos definir estados y queremos que nuestro código sea más fácil de entender y mantener.

```
typedef enum animo {CANSADO, ANIMADO, NORMAL} Animo;
typedef struct Humano {
  int  edad;
  Animo estado;
} Persona;

Persona p;
p.edad = 21;
p.estado = CANSADO;
```

Con `enum` también podemos usar `typedef` como si fueran otro tipo de dato más.

ENUMS O ENUMERADORES

Otro ejemplo:

```c
#include <stdio.h>

enum semana{Lun, Mar, Mie, Jue, Vie, Sab, Dom};

int main()
{
    enum semana dia;
    dia = Mie;
    printf("%d", dia);
    return 0;
}
```

Output: 2

Compilación con Makefile

COMPILACIÓN CON MAKEFILE

El `Makefile` es un archivo muy especial que sirve para compilar programas grandes, en los que existe una estructura más compleja de directorios y archivos.

Antes de usarlo, deberás situarlo en el directorio raíz de tu proyecto y configurarlo acorde a eso. Para ejecutar la compilación basta con escribir el comando `make` en consola desde el mismo directorio.

Errores de memoria y Valgrind

ERRORES DE MEMORIA

A veces, puede ocurrir que nuestro programa se caiga y lo único que nos diga el computador sea "Segmentation Fault (core dumped)". ¿Que sucedió?

Un error de segmentación o `segfault` es levantado por nuestro sistema operativo cuando intentamos acceder a memoria que no nos corresponde, posiblemente debido a un acceso a un puntero indebido.

Ejemplos

- Intentar acceder a la memoria de un puntero NULL.
- Acceder a un puntero previamente liberado con free().
- Intentar liberar el mismo puntero dos veces.

Frente a esto, el sistema operativo le envía una señal `SIGSEGV` al proceso, terminándolo. Esta señal es capturable, sin embargo, NO se recomienda hacerlo.

VALGRIND

Cuando encuentras errores de este tipo, ¿cómo saber qué parte del código produjo el error?

Existen múltiples herramientas para ver el uso de memoria, pero en este curso usaremos `valgrind`. Para instalarlo en Linux solo deben correr el comando:

```
sudo apt-get install valgrind
```

Para usar `valgrind`, basta con anteponer `valgrind` antes del ejecutable y sus parámetros:

```
valgrind ./ejecutable <parámetro1> .. <parámetroN>
```

Para instalarlo en MAC pueden seguir este link.

Para Windows 10 deben instalar y ejecutar la linux bash shell y luego correr el comando indicado.

OUTPUTS DE VALGRIND

En `valgrind` puedes recibir múltiples salidas.

```
==11321==
==11321== HEAP SUMMARY:
==11321==     in use at exit: 0 bytes in 0 blocks
==11321==   total heap usage: 11 allocs, 11 frees, 1,154 bytes allocated
==11321==
==11321== All heap blocks were freed -- no leaks are possible
==11321==
==11321== For counts of detected and suppressed errors, rerun with: -v
==11321== ERROR SUMMARY: 0 errors from 0 contexts (suppressed: 0 from 0)
```

Figura: Manejo de memoria perfecto.

En este caso, liberamos toda la memoria pedida, sin excepciones y no tuvimos errores de memoria.

Se dice que no existen `memory leaks`.

OUTPUT DE VALGRIND

Tambien se puede dar el caso en el que no se libera toda la memoria pedida, ya sea porque simplemente se te olvidó liberarla o“se perdió”, es decir, que no es alcanzable.

```
==11379==
==11379== HEAP SUMMARY:
==11379==     in use at exit: 66 bytes in 6 blocks
==11379==   total heap usage: 11 allocs, 5 frees, 1,154 bytes allocated
==11379==
==11379== LEAK SUMMARY:
==11379==    definitely lost: 56 bytes in 5 blocks
==11379==    indirectly lost: 10 bytes in 1 blocks
==11379==      possibly lost: 0 bytes in 0 blocks
==11379==    still reachable: 0 bytes in 0 blocks
==11379==         suppressed: 0 bytes in 0 blocks
==11379== Rerun with --leak-check=full to see details of leaked memory
==11379==
==11379== For counts of detected and suppressed errors, rerun with: -v
==11379== ERROR SUMMARY: 0 errors from 0 contexts (suppressed: 0 from 0)
```

Figura: Manejo de memoria con `memory leaks`.

En este caso, no pudimos liberar toda la memoria pedida, tenemos `memory leaks`.

OUTPUT DE VALGRIND

Finalmente, podemos observar errores de memoria más graves, como cuando se leen variables que no han sido inicializadas (aún no les damos un valor). Con suerte, estos errores no presentan un mayor problema para la ejecución. Sin embargo, significan que algo malo está pasando. Errores más graves pueden conllevar a una `segfault`.

```
==11427== HEAP SUMMARY:
==11427==     in use at exit: 130 bytes in 10 blocks
==11427==   total heap usage: 11 allocs, 1 frees, 1,154 bytes allocated
==11427==
==11427== LEAK SUMMARY:
==11427==    definitely lost: 0 bytes in 0 blocks
==11427==    indirectly lost: 0 bytes in 0 blocks
==11427==      possibly lost: 0 bytes in 0 blocks
==11427==    still reachable: 130 bytes in 10 blocks
==11427==         suppressed: 0 bytes in 0 blocks
==11427== Rerun with --leak-check=full to see details of leaked memory
==11427==
==11427== For counts of detected and suppressed errors, rerun with: -v
==11427== Use --track-origins=yes to see where uninitialised values come from
==11427== ERROR SUMMARY: 28 errors from 8 contexts (suppressed: 0 from 0)
Segmentation fault (core dumped)
ichottmano@ichottmano:~/Documents/ayudantía$
```

Figura: Manejo de memoria con errores y una `segfault`.

OUTPUT DE VALGRIND

En este último caso, `valgrind` nos permite ver en qué parte del código ocurren los errores de memoria, mostrando las funciones que tuvieron problemas.

```
==11427==
==11427== Process terminating with default action of signal 11 (SIGSEGV)
==11427==  General Protection Fault
==11427==    at 0x4C32CF2: strlen (in /usr/lib/valgrind/vgpreload_memcheck-amd64
-linux.so)
==11427==    by 0x4E994D2: vfprintf (vfprintf.c:1643)
==11427==    by 0x4EA0F25: printf (printf.c:33)
==11427==    by 0x10889D: saludar (in /home/ichottmano/Documents/ayudantía/mem)
==11427==    by 0x10896F: main (in /home/ichottmano/Documents/ayudantía/mem)
```

Figura: Salida de `valgrind` donde observamos que la `segfault` fue causada en la función “saludar”, llamada por el `main` de nuestro código.

errno.h

ERRNO.H

`errno.h` es un header file de C útil para identificar algunos errores. Al usarlo, se define la variable `errno` de valor entero (`int`) con valor inicial `0`. Llamadas al sistema o system calls y algunas funciones pueden cambiar el valor de esta variable cuando incurren en algún error.

Como se trata de un header file, para usarlo se debe incluir en el programa con:

```
#include <errno.h>
```

Su uso es práctico cuando, en el caso de errores, alguna función o llamada al sistema retorna `-1` o `NULL`, lo que no es muy descriptivo.

```c
#include <stdio.h>
#include <errno.h> //Incluimos errno.h a nuestro programa
#include <string.h>

int main()
{
  FILE *fp;

  /*
  Si abrimos un archivo que no existe,
  obtendremos un error
  */
  fp = fopen("YoNoExisto.txt", "r");

  printf("Valor de errno: %d\n", errno);
  printf("El mensaje de error es: %s\n", strerror(errno));
  perror("Mensaje del error");

  return 0;
}
```

Figura: Ejemplo de uso de `errno.h` en donde se abre un archivo que no existe, causando un error.

```
Valor de errno: 2
El mensaje de error es: No such file or directory
Mensaje del error: No such file or directory
```

Figura: Output del ejemplo anterior. Se aprecia el valor de `errno` una vez ocurrido el error y su significado.

Para más información pueden revisar este link.

USO DEL SERVIDOR - SSH Y SCP

USO DEL SERVIDOR - SSH Y SCP

A lo largo del curso tendrán que subir sus trabajos diréctamente al servidor del curso. Para esto, usaremos **SSH** que nos permite ejecutar comandos en el servidor de forma remota.

El comando a ejecutar desde consola es el siguiente:

`ssh <username>@iic2333.ing.puc.cl`

Donde <username> es su correo UC (sin '@uc.cl'). La clave de todos es su **`número de alumno`**. Si desean cambiarla pueden hacerlo, una vez iniciada la sesión, ejecutando el comando **`passwd`**.

USO DEL SERVIDOR - SSH Y SCP

Cada alumno del curso tiene un directorio propio en el servidor. Para subir archivos al servidor, deben usar el comando `scp` de la siguiente forma:

- Para subir archivos:
  `sudo scp Desktop/file.txt <username>@iic2333.ing.puc.cl:/user/<username>/Ti`
- Para subir carpetas:
  `sudo scp -r Desktop/myfolder <username>@iic2333.ing.puc.cl:/user/<username>/Ti`

Asumiendo que deja todos sus archivos en la carpeta `Desktop` y `Ti` la carpeta de su tarea en el servidor. (Esta última de ejemplo `debe` existir dentro de su directorio del servidor si quiere probar el comando.)

USO DEL SERVIDOR - SSH Y SCP

Algunas consideraciones:

- `sudo` hace referencia a 'super user do', lo que nos permite ejecutar comandos con privilegios de seguridad.
- El comando `scp` hace referencia a 'secure copy' o copia segura. Este es el que nos permite hacer copias desde y hacia el servidor a través de nuestro computador local.
- Recuerden que una vez que acceden remotamente al servidor, no tendrán acceso a sus archivos locales. Es decir, `scp` no funcionará para transferir archivos locales al servidor mientras estén conectados a este último.

Uso del servidor - SSH y SCP (en Windows)

En Windows para conectarse al servidor pueden usar el programa PuTTY. Para transferir archivos via PuTTY deben instalar PSCP.

Otra alternativa es usar FileZilla, que provee una interfaz más cómoda, para mover archivos entre el servidor y su computador local.

En ambos casos deben ingresar el nombre del servidor, su nombre de usuario y contraseña para conectarse.

USO DEL SERVIDOR - SSH Y SCP (EN WINDOWS)

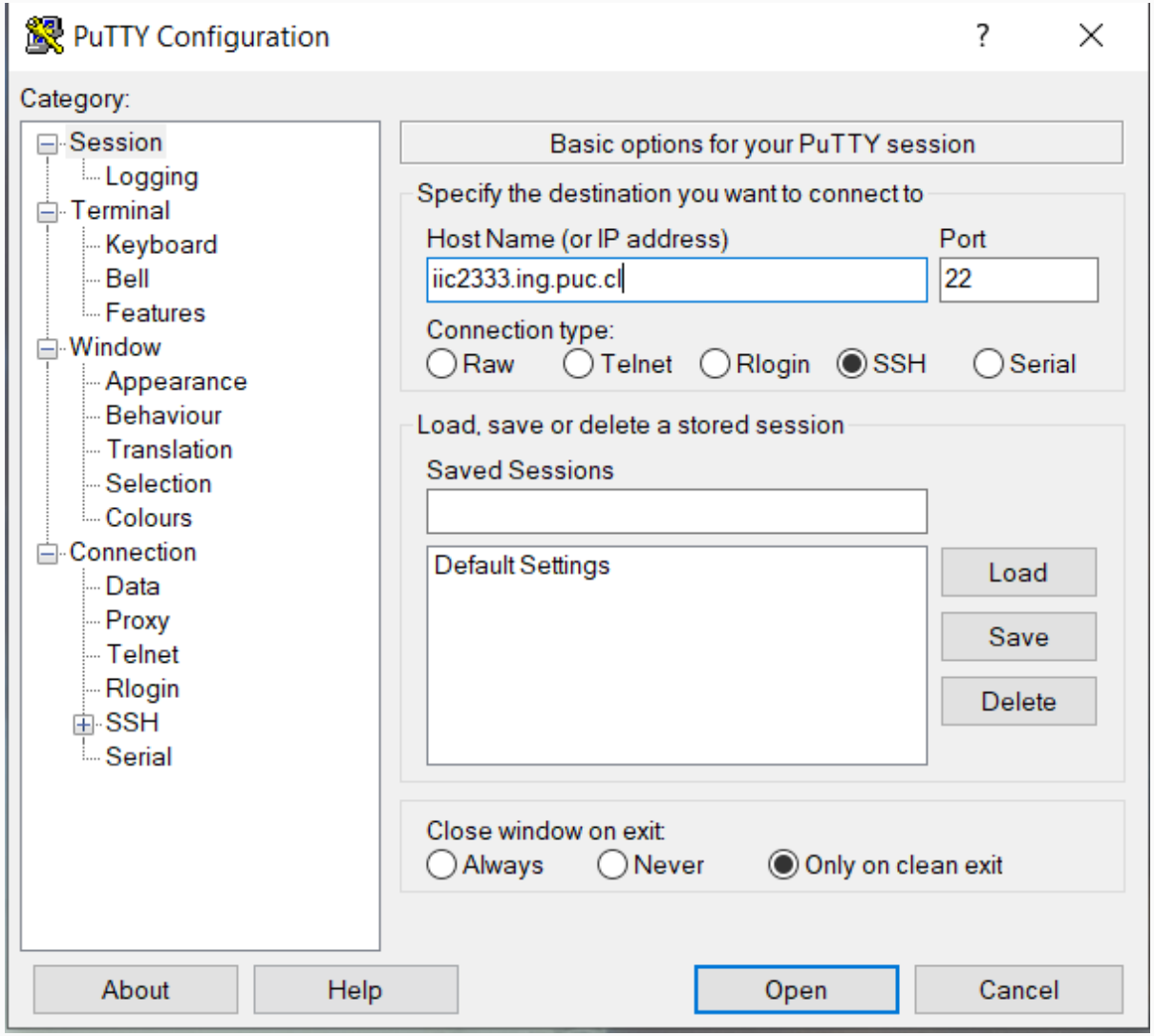

Figura: Ejemplo de uso de PuTTY. Deben marcar **SSH** como tipo de conexión. Al hacer click en **Open** se les preguntará por su <username> y contraseña.

ENLACES Y RECURSOS ÚTILES

ENLACES Y RECURSOS ÚTILES

- Learn C
- w3schools
- TutorialsPoint - Librerías en C
- The Geek Stuff - Ejemplo de señales
- The Linux Man Pages Project

FIN (/°U°)/