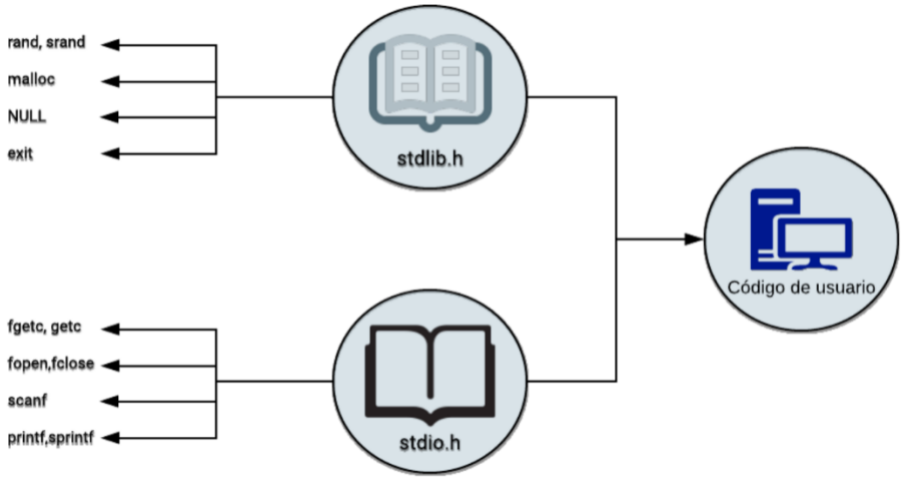
Librerías y comandos *make*

Librerías

Las librerías son un grupo de archivos que tienen una funcionalidad pre-construida, y que pueden ser usadas por cualquier ejecutable. Las librerías contienen en su interior variables y funciones.



Estos archivos contienen las especificaciones de diferentes funcionalidades que, al poder incluirlas en el código, permitirá ahorrar gran cantidad de desarrollo y tiempo.

Cómo crear librerías propias

Utilizar librería dentro del mismo directorio

Por ejemplo, se quiere crear una librería para las funciones del IPC de Linux y se desea incluir esta librería para probarla en otro archivo. Para esto se siguen los siguientes pasos:

1) Se crea el archivo de cabecera con extensión ".h" en el mismo directorio del código principal, éste archivo debe tener todos los prototipos de funciones y definiciones de tipos de datos de la librería propia:

```
#ifndef _FUNCIONES_H
#define _FUNCIONES_H

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

```h
#include <stdlib.h>
#include <sys/ipc.h>
#include <sys/sem.h>
#include <sys/shm.h>
#include <sys/msg.h>
#include <time.h>
#include <string.h>

#define CLAVE_BASE 33
#define SEMAFORO0 0

#define LARGO_TMENSAJE 1024


typedef struct tipo_mensajes mensaje;
struct tipo_mensajes
{
    long    long_dest;                  //Destinatario
    int     int_rte;                    //Remitente
    int     int_evento;                 //Numero de nevento
    char    char_mensaje[LARGO_TMENSAJE];   //mensajse
};

key_t   creo_clave(int);
void*   creo_memoria(int, int*, int);
int     creo_semaforo(int);
void    inicio_semaforo(int, int, int);
void    levanta_semaforo(int, int);
void    espera_semaforo(int, int);
int     creo_id_cola_mensajes(int);
int     borrar_mensajes(int);
int     recibir_mensaje(int, long , mensaje*);
int     enviar_mensaje(int, long, int, int, char*);

#endif
```

2) Se crea el archivo del código de la librería; contiene el archivo de cabecera creado anteriormente y además incluye el código de todas las funciones que fueron escritas en el archivo de cabecera:

```h
#include "funciones.h"

key_t creo_clave(int r_clave)
{
```

```
    key_t clave;
    clave = ftok ("/bin/ls", r_clave);
    if (clave == (key_t)-1)
    {
        printf("No puedo conseguir clave para memoria compartida\n");
        exit(0);
    }
    return clave;
}
void* creo_memoria(int size, int* r_id_memoria, int clave_base)
{
    void* ptr_memoria;
    int id_memoria;
    id_memoria = shmget (creo_clave(clave_base), size, 0777 | IPC_CREAT);

    if (id_memoria == -1)
    {
        printf("No consigo id para memoria compartida\n");
        exit (0);
    }

    ptr_memoria = (void *)shmat (id_memoria, (char *)0, 0);

    if (ptr_memoria == NULL)
    {
        printf("No consigo memoria compartida\n");

        exit (0);
    }
    *r_id_memoria = id_memoria;
    return ptr_memoria;
}

int creo_semaforo(int cuantos)
{
  key_t clave = creo_clave(CLAVE_BASE);
  int id_semaforo = semget(clave,cuantos,0600|IPC_CREAT);
  if(id_semaforo == -1)
  {
      printf("Error: no puedo crear semaforo\n");
      exit(0);
  }
  return id_semaforo;
}
```

```c
void inicio_semaforo(int id_semaforo, int cual,int valor)
{
    semctl(id_semaforo,cual,SETVAL,valor);
}

void levanta_semaforo(int id_semaforo,int cual)
{
    struct sembuf operacion;
    operacion.sem_num = cual;
    operacion.sem_op = 1; //incrementa el semaforo en 1
    operacion.sem_flg = 0;
    semop(id_semaforo,&operacion,1);
}

void espera_semaforo(int id_semaforo,int cual)
{
    struct sembuf operacion;
    operacion.sem_num = cual;
    operacion.sem_op = -1; //decrementa el semaforo en 1
    operacion.sem_flg = 0;
    semop(id_semaforo,&operacion,1);

}


int creo_id_cola_mensajes(int clave)
{
    int id_cola_mensajes = msgget (creo_clave(clave), 0600 | IPC_CREAT);
    if (id_cola_mensajes == -1)
    {
        printf("Error al obtener identificador para cola mensajes\n");
        exit (-1);
    }
    return id_cola_mensajes;
}

int enviar_mensaje(int id_cola_mensajes, long rLongDest, int rIntRte, int rI
ntEvento, char* rpCharMsg)
{
    mensaje msg;
    msg.long_dest   = rLongDest;
    msg.int_rte     = rIntRte;
    msg.int_evento  = rIntEvento;
    strcpy(msg.char_mensaje, rpCharMsg);
```

```
    return msgsnd (id_cola_mensajes, (struct msgbuf *)&msg, sizeof(msg.int_r
te)+sizeof(msg.int_evento)+sizeof(msg.char_mensaje), IPC_NOWAIT);
}

int recibir_mensaje(int id_cola_mensajes, long rLongDest, mensaje* rMsg)
{
    mensaje msg;
    int res;
    res = msgrcv (id_cola_mensajes, (struct msgbuf *)&msg,  sizeof(msg.int_r
te)+sizeof(msg.int_evento)+sizeof(msg.char_mensaje),    rLongDest, 0);

    rMsg->long_dest         = msg.long_dest;
    rMsg->int_rte       = msg.int_rte;
    rMsg->int_evento    = msg.int_evento;
    strcpy(rMsg->char_mensaje, msg.char_mensaje);
    return res;
}

int borrar_mensajes(int id_cola_mensajes)
{
    mensaje msg;
    int res;
    do
    {
        res = msgrcv (id_cola_mensajes, (struct msgbuf *)&msg,  sizeof(msg.i
nt_rte)+sizeof(msg.int_evento)+sizeof(msg.char_mensaje),    0, IPC_NOWAIT);
    }while(res>0);
    return res;
}
```

3) Se crea el archivo “definiciones.h” para todas las definiciones comunes puntuales de los ejercicios. Como ejemplo, se va a usar un ejercicio con 2 procesos: Supermercado y Repositor, que corren en el SO Linux:

```
#ifndef _DEFINICIONES_H
#define _DEFINICIONES_H

#define LARGO_DESCRIPCION 100
#define CANT_SEMAFORO 1

typedef struct tipo_datos datos;
struct tipo_datos
{
    int dato;
```

```
};

typedef enum
{
    MSG_NADIE,
    MSG_SUPERMERCADO,
    MSG_REPOSITOR
}Destinos;

typedef enum
{
    EVT_NINGUNO,
    EVT_CONSULTA_STOCK,
    EVT_RESPUESTA_STOCK,
    EVT_SUMA_STOCK,
    EVT_RESTA_STOCK
}Eventos;

#endif
```

4) Llamar a la librería. Cuando estén terminados ambos archivos, la librería está lista para ser usada. Se crean los archivos supermercado.c y repositor.c, dentro del mismo directorio, se incluyen las nuevas librerías creadas:

repositor.c :

```
#include "funciones.h"
#include "definiciones.h"
#define INTERVALO_PEDIDOS 2000

void procesar_evento(int id_cola_mensajes, mensaje msg)
{
    printf("Destino   %d\n", (int) msg.long_dest);
    printf("Remitente %d\n", msg.int_rte);
    printf("Evento    %d\n", msg.int_evento);
    printf("Mensaje   %s\n", msg.char_mensaje);
    switch (msg.int_evento)
    {
        case EVT_RESPUESTA_STOCK:
            printf("Rta stock\n");
            printf("STOCK %d\n", atoi(msg.char_mensaje));
        break;

        default:
            printf("\nEvento sin definir\n");
        break;
```

```
    }
    printf("-----------------------------\n");
}

int main()
{
    int             id_cola_mensajes;
    mensaje         msg;
    id_cola_mensajes    = creo_id_cola_mensajes(CLAVE_BASE);
    while(1)
    {
        enviar_mensaje(id_cola_mensajes , MSG_SUPERMERCADO, MSG_REPOSITOR, E
VT_SUMA_STOCK, "SUMA UNO");
        enviar_mensaje(id_cola_mensajes , MSG_SUPERMERCADO, MSG_REPOSITOR, E
VT_CONSULTA_STOCK, "DECIME EL STOCK POR FAVOR");
        recibir_mensaje(id_cola_mensajes, MSG_REPOSITOR, &msg);
        procesar_evento(id_cola_mensajes, msg);
        usleep (INTERVALO_PEDIDOS*1000);
    };

    return 0;
}
```

supermercado.c :

```
#include "funciones.h"
#include "definiciones.h"
#define INTERVALO_PEDIDOS 2000

void procesar_evento(int id_cola_mensajes, mensaje msg)
{
    char cadena[100];
    //cantidad es inicializada solo una vez,
    //y mantiene el valor en las sucesivas llamadas.
    static int cantidad=0;
    printf("Destino   %d\n", (int) msg.long_dest);
    printf("Remitente %d\n", msg.int_rte);
    printf("Evento    %d\n", msg.int_evento);
    printf("Mensaje   %s\n", msg.char_mensaje);
    switch (msg.int_evento)
    {
        case EVT_CONSULTA_STOCK:
            printf("Consulta Stock\n");
            sprintf(cadena, "%d", cantidad);
```

```
            enviar_mensaje(id_cola_mensajes , msg.int_rte, MSG_SUPERMERCADO,
 EVT_RESPUESTA_STOCK, cadena);
        break;
        case EVT_SUMA_STOCK:
            printf("Suma Stock\n");
            cantidad++;
            break;
        default:
            printf("\nEvento sin definir\n");
        break;
    }
    printf("-----------------------------\n");
}
int main()
{
    int             id_cola_mensajes;
    mensaje          msg;
    id_cola_mensajes    = creo_id_cola_mensajes(CLAVE_BASE);
    while(1)
    {
        recibir_mensaje(id_cola_mensajes, MSG_SUPERMERCADO, &msg);
        procesar_evento(id_cola_mensajes, msg);
    };

    return 0;
}
```

Observar que las librerías se incluyen entre comillas “” y no con los signos <>, esto es así para que as busque en el mismo directorio o ruta indicada. Cuando tiene los signos <> el compilador busca los archivos en el directorio donde están instaladas las librerías estándar del lenguaje C.

5) Para compilar estos procesos con las nuevas librerías hay que escribir por consola lo siguiente:

```
gcc -o super supermercado.c funciones.c
gcc -o rep repositor.c funciones.c
```

Estas líneas enlazan el archivo “funciones.c” que contiene las funciones de código de la librería “funciones.h” (incluida en los archivos del código fuente de supermercado y repositor).

Comando *make*

Make es una herramienta de gestión de dependencias usada en entornos Linux. Cuando el código se compone de varios archivos de código fuente, es muy práctica para compilar de una forma automatizada todo lo que necesitas y pasarle a GCC las opciones necesarias. Además, *make* sabrá qué cosas hay que recompilar en caso de que hagas cambios, algo bastante práctico cuando el proyecto es grande.

Además de eso, *make* entiende los formatos o extensiones. Por ejemplo, si es un hola.c sabe que le debe encargar la tarea a gcc, mientras que si es un hola.cpp se lo encomendará a g++, etc.

La utilidad *make* requiere de un archivo, *makefile*, que define un conjunto de tareas a ejecutar y describe como compilar un programa.

Archivo makefile

Un archivo *makefile* básicamente se distingue en cuatro tipos básicos de declaraciones:

- Comentarios
- Variables.
- Reglas explicitas.
- Reglas implícitas.

Las **reglas explícitas** le indican a *make* que archivos dependen de otros archivos, así como los comandos requeridos para compilar un archivo en particular.

Las **reglas implícitas** son similares que las explicitas, pero con la diferencia que indican los comandos a ejecutar, sino que *make* utiliza las extensiones de los archivos para determinar que comandos ejecutar.

Cree un archivo *makefile* con el siguiente contenido:

```
# Esto es un comentario.

CC      = g++

all : repositor supermercado

supermercado : supermercado.c funciones.h definiciones.h funciones.o
    cc -o supermercado supermercado.c funciones.o
```

```
repositor : repositor.c funciones.h definiciones.h funciones.o
    cc -o repositor repositor.c funciones.o

funciones.o : funciones.c funciones.h
    cc -c funciones.c

clean:
    rm -rf *o supermercado repositor
```

Los renglones iniciados con # son comentarios, no se ejecutan. El resto de las líneas son reglas. Cada regla empieza con un nombre seguido de ":". Las líneas iniciadas con tabulador son continuación de la misma regla.

El archivo *makefile* debe ubicarse en el mismo directorio del código fuente.

Compilar el programa dando el comando:

$ make

Observar los mensajes indicativos de las tareas realizadas. Luego ejecutar el código, para verificar su funcionamiento:

$ ./repositor

Ejecutando la siguiente sentencia, se borra el ejecutable y los archivos objeto:

$ makeclean

Reglas de *make*

Es posible hacer que *make* sea más listo escribiendo un fichero, uno de cuyos nombres por defecto es *makefile*, y decirle en él qué cosas debe hacer y cómo.

En ese fichero se incluirán cosas como:

objetivo: **dependencia1dependencia2** ...

<tab>**comando1**

<tab>**comando2**

“**objetivo**” es lo que se desea construir. Puede ser el nombre de un ejecutable, el nombre de una librería o cualquier palabra inventada. Para este ejemplo, se usa: all, supermercado, repositor, funciones.o y *clean*.

dependencia<i> es el nombre de otro objetivo que debe hacerse antes que el propio, o bien ficheros de los que depende el objetivo. En el ejemplo, las dependencias serían los ficheros fuente, ya que para hacer el ejecutable se necesitan todos esos fuentes y, si se los toca, posiblemente se deberá rehacer el ejecutable.

<tab> es un tabulador. Es importante que ahí se ponga un tabulador, porque sino el fichero no se lee correctamente.

comando<i> es lo que se tiene que ejecutar para construir el objetivo. Se irán ejecutando estos comandos en secuencia. Puede ser cualquier comando de shell válido. (cc, rm, cp, ls, o incluso un script que se haya hecho). En el ejemplo, sería:

```
cc -o supermercado supermercado.c funciones.o
cc -o repositor repositor.c funciones.o
cc -c funciones.c
rm -rf *o supermercado repositor
```