Librerías

Algo importante en el desarrollo de un programa es la reutilización del software, la reutilización del software se logra mediante la utilización de librerías. Una librería es una compilación de funciones que han creado otros programadores o uno mismo. Esta está compuesta por un archivo de declaraciones (.h, .hh, .hpp) y por el archivo compilado de la implementación de la librería (.o, .so. dll), generado a partir del código fuente (.c, .cpp).

El archivo de declaraciones contiene la definición de funciones, clases, métodos que se han implementado en la librería, esto es lo único que necesita conocer el programador, qué tiene la librería, no como se implementó. Esto representa modularización y encapsulamiento.

Para incluir una librería se hace uso de la directiva #include "__miLibrería__" o #include <__libreríaDelSistema__>. Cuando una librería es del sistema (está en el *path* del compilador) se coloca entre <>, cuando es creada por el usuario (está en la carpeta del proyecto) se coloca entre comillas dobles (" ").

A diferencia de C las librerías en C++ no tienen extensión. Asimismo, para los fanáticos de C, las librerías que se usaban en C están disponibles en C++, para ello se antepone una c antes del nombre de la librería y se quita la extensión, por ejemplo en C se tenía:

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

Las correspondientes librerías en C++ serían

```
#include <cstdio>
#include <cstdlib>
```

Otra novedad en C++ es la introducción de los espacios de nombres (namespaces), esto se hizo para poder utilizar los mismos identificadores en una misma librería, lo que se hace es incluir la definición de la librería dentro de un espacio de nombres, en el curso siempre se utilizará el espacio de nombres estándar, para ello se debe incluir la siguiente sentencia, después de la declaración de librerías:

```
using namespace std;
```

¡Mola Hundo!

Este es el primer programa que típicamente se hace al aprender un lenguaje de programación (o al menos su version sin copyright ;). El siguiente código muestra el programa en C++:

```
#include<iostream>
int main(void){
  std::cout<<"Mola Hundo!" <<endl;
  return 0;
}
```

De este pequeño programa se pueden destacar varias cosas, en primer lugar la función *int main(void)* todo programa ejecutable en C o C++ debe tener una función main, esta es la función que se ejecutará siempre al inicio de un programa.

Está función no toma parámetros (se pueden hacer versiones que tomen parámetros al inicio), y debe retornar un int al final de su ejecución, si retorna cero es un indicador para el sistema operativo que el programa terminó satisfactoriamente, algo diferente de cero sirve para indicar una condición de error.

Lo segundo a notar es la librería que se incluye: *iostream*, la cual contiene definiciones de clases que sirven para comunicarse con el estándar input (*cin*: usualment e el teclado), y con el estándar output (*cout*: usualmente el monitor), también contiene métodos para darle formato a los datos.

El objeto *cout* sirve para enviarle cosas al estándar output las cosas se les envían con <<, de esta forma se pueden enviar strings, variables, constantes y otros. El objeto *endl* es el indicador de fin de línea.

Antes de utilizar el objeto *cout* se utiliza el prefijo *std::*, este sirve para indicarle al compilador que el objeto *cout* que se desea utilizar, es el que está en el espacio de nombres estándar. Se le puede decir al compilador que el espacio de nombres estándar es el que siempre se quiere usar, esto se puede ver en la siguiente versión del programa ¡Mola Hundo!.

mola.cpp:

```
#include<iostream>

using namespace std;

int main(void){
  cout<<"Mola Hundo!" <<endl;
  return 0;
}
```

Para crear un archivo ejecutable del programa anterior se debe llamar al compilador de C++, el compilador GNU de C++ se llama *g++*, supóngase que el programa anterior se salvó con el nombre de mola.cpp, para crear un programa ejecutable se debe de ejecutar el siguiente comando:

```
g++ -o mola mola.cpp
```

La línea anterior llama al compilador con la bandera *o*, esta bandera es para que cree un ejecutable, el siguiente argumento es el nombre que se desea poner al ejecutable, y el último argumento es el nombre del archivo fuente. Note que en Linux un ejecutable no necesita tener una extensión definida, (por ejemplo .exe), sólo se le debe poner la bandera de ejecución.

Para ejecutar el programa sólo se debe ejecutar esto

```
./mola
```

- Copie el programa Mola Hundo!
- Cree el ejecutable mola.
- Corra el programa mola.

Responda:

1. ¿Porqué se llama el programa de esta manera y no simplemente digitando *hola*?
2. ¿Qué debe hacerse para poder correrlo escribiendo simplemente *hola*?
3. ¿Porqué el devolver un 0 es signo de ejecución correcta y un valor distinto indica una condición de error?

Control de versiones

Antes de continuar, inicialice un repositorio git en su directorio de trabajo actual, agregue los archivos fuente que se han creado hasta el momento, realice un add y un commit inicial, junto con un comentario de documentación para el repositorio. No olvide configurar el nombre del creador del repositorio y su correo electrónico.

Cree y configure el archivo .gitignore para no incluir en el repositorio los archivos temporales ni los generados automáticamente, luego de la compilación.

Responda:

1. Describa las reglas de archivo *.gitignore* para ignorar archivos específicos y wild cards.
2. ¿Porqué el archivo *.gitignore* no se agregó al index del repositorio. ¿Qué se debe hacer para lograrlo?

Realice un clone del repositorio del compañero que se encuentra a su derecha, en un nuevo directorio. Compile el código del compañero en el working directory y ejecute el mola.

Funciones y compilación de fuentes múltiples

Cuando se trabaja con proyectos grandes lo mejor es dividir el proyecto en múltiples fuentes, cada una con una funcionalidad particular.

A continuación se hará el programa Mola Hundo!, en una función que se implementará en otro archivo.

Para hacer esto se tendrán dos archivos, uno que se llame mola.hh y otro que se llame mola.cpp, en estos archivos se tendrá lo siguiente:
mola.hh:

```
bool mola(void);
```

mola.cpp:

```
#include<iostream>
using namespace std;
bool mola(void){
  cout<<"Mola Hundo!" <<endl;
  return true;
}
```

Por último se tendrá un programa principal que hará uso de la función hola:
main.cpp:

```
#include "mola.hh"
int main(void){
  mola();
```

```
  return 0;
}
```

Cuando se tienen múltiples fuentes, se debe compilar primero todas las fuentes que utiliza la función main, y después (mediante el enlazado) crear el archivo ejecutable.

Para compilar una fuente se hace llamando al compilador con la bandera *c*. Esta bandera le indica al compilador que sólo compile, que no debe crear un archivo ejecutable. Al compilar un archivo, se creara un archivo con el mismo nombre de la fuente pero con extensión *.o*, este archivo contiene el código de máquina del programa a compilar, y una tabla con las funciones externas que están siendo usadas, en este caso las relacionadas con *cout*. Para compilar *mola.cpp* se ejecuta:

```
g++ -c mola.cpp
```

Ahora para crear el archivo ejecutable se hace algo similar a lo que se hacía anteriormente, cuando no se tenían múltiples fuentes:

```
g++ -o mola main.cpp mola.o
```

Con este comando se le dice al compilador, cree el archivo ejecutable *mola*, tomando la función main de *main.cpp* y uniéndolo a *mola.o*. Al hacer esto el compilador compila *main.cpp* y se çonvierte.[en] un enlazador (linker), el cual une todas las dependencias que existen, y crea el ejecutable.

- Cree los archivos mola.hh, mola.cpp, main.cpp
- Compile mola.cpp y asegúrese que se creo el archivo mola.o, ábralo y describa qué hay dentro.
- Cree el ejecutable mola
- Ejecute mola
- Agregue los nuevos archivos al repositorio

Makefiles

Como se vio en la sección anterior cuando se tienen múltiples fuentes es necesario compilar cada una de ellas en forma independiente. Imagínese un trabajo el que se tienen 20 fuentes diferentes, es muy tedioso tener que compilar cada uno de ellos.

Aunque después de hacer una cambio en una fuente se pueden recompilar todos los archivos sin importar si cambiaron o no, esto es ineficiente dado que se están utilizando recursos de la computadora en vano, además en proyectos grandes compilar un archivo puede tomar minutos u horas. Para remediar esto se deben compilar sólo las fuentes que cambian. Al compilar las fuentes que cambian hay que tener cuidado con las dependencias, por ejemplo si se tienen una fuente A que incluye cosas de B, y B incluye cosas de C. Si sólo se modifica A sólo hay que recompilar A, no hay que compilar ni B ni C. Si se cambia B hay que recompilar B y A(dado que depende de B) pero no C, si se cambia C hay que recompilar las tres fuentes.

Una forma de solucionar el problema de las fuentes que cambian, no recompilando todo cada vez, y manteniendo las dependencias es hacer uso de un makefile.

Para manejar múltiples fuentes lo ideal es trabajar en proyectos, un proyecto es una forma de agrupar los archivos fuentes, y de crear reglas para su compilación. En Linux tradicionalmente lo usado para trabajar con

proyectos es el comando make este comando lee lo que existe en un archivo llamado makefile y siguiendo las reglas ahí descritas compila el proyecto.

El makefile para compilar el programa mola hundo! original sería:

```
#Esto es un comentario
#Se definen los objetos, estos son los archivos que necesitan
#para crear el ejecutable.
OBJS = main.cpp

#Se define el compilador
CC = g++

#Bandera de depuracion
DEBUG = -g

#Banderas de compilacion
CFLAGS = -Wall -c $(DEBUG) -pedantic

#Banderas para el Linker
LFLAGS = -Wall $(DEBUG) -pedantic

#Archivo Ejecutable que se va a crear
TARGET = main

#Reglas a ejecutar cuando se ejecute make
all:
$(TARGET) : $(OBJS)
$(CC) $(LFLAGS) $(OBJS) -o $(TARGET)

#Al digitar make clean se borraran todos los archivos compilados,
#note el -f en rm, si no sabe para que sirve use el manual de rm.
clean:
rm -f *.o $(TARGET)
```

Para el caso del Mola Hundo! con mútiples fuentes el makefile sería (tenga mucho cuidado si va a copiar y pegar el texto del makefile):

```
OBJS = main.cpp mola.o
CC = g++
DEBUG = -g
CFLAGS = -Wall $(DEBUG) --pedantic -c
LFLAGS = -Wall --pedantic $(DEBUG)
TARGET = main
$(TARGET) : $(OBJS)
$(CC) $(LFLAGS) $(OBJS) -o $(TARGET)
mola.o: mola.h mola.cpp
$(CC) $(CFLAGS) mola.cpp
clean:
rm -f *.o $(TARGET)
```

Responda:

1. Escriba el makefile para ambos ejemplos y compruebe que funcionan, para correr un makefile basta con digitar make, al hacer esto se ejecutará lo que está en la etiqueta all, para borrar los archivos creados se debe digitar make clean, con esto se ejecutará la parte de borrar. Pruebe make clean utilice ls para ver que funciona.
2. Compruebe que los programs generados funcionan.
3. ¿Porqué se debe tener cuidado a la hora de copiar el texto del makefile?
4. Actualice su repositorio
5. Actualice la copia del repositorio de su compañero localmente.

Punteros

Un puntero es una posición de memoria que se utiliza para guardar la dirección de un dato.

En forma muy básica un programa posee tres tipos de memoria: Variables, Heap, Pila.
Todas las variables que se crean, lo hacen en el área de memoria reservado para ello, asimismo, allí se crean los punteros. Un puntero puede señalar una posición de memoria de cualquiera de los tres bloques. El heap se utiliza para la asignación de memoria dinámica, por último la pila es el área de almacenamiento temporal. Cuando se llama una función, sus parámetros se pasan por la pila, de forma que al pasar una variable o puntero a una función se pasa una copia de estos (esto es, mediante paso por valor).

Para entender la necesidad de los punteros se presentan los siguientes ejemplos, asimismo, en ellos se presentan los diferentes tipos de punteros que existen.

Qué pasa cuando se pasa una variable directamente a una función y trata de modificarse su valor?

```
#include <iostream>
using namespace std;

void square(int n){
  n = n*n;
  cout<<"Segun la funcion void square(int n) el valor de numero es: "<< n <<endl;
}

void square(int *ptr_n){
  *ptr_n = *ptr_n * *ptr_n;
  cout << "Segun la funcion void square(int *ptr_n) el valor
 de numero es" << *ptr_n << endl;
}

int main(void){
  int numero = 10, *ptr_num;
  //Paso de parametros por valor
  cout<<"El valor de la variable numero antes de la funcion void square(int)
  es:  "<<numero<<endl;
  square(numero);
  cout<<"El valor de la variable numero despues de la funcion void square(int)
```

```
  es: "<<numero<<endl;

  //Paso de parametros por referencia
  cout<<"\nEl valor de la variable numero antes de la funcion void square(*int)
  es: "<<numero<<endl;
  square(&numero);
  cout<<"El valor de la variable numero despues de la funcion void square(*int)
  es:"<<numero<<endl;

  //Se puede manejar con punteros
  ptr_num=&numero;
  cout<<"\nEl valor de la variable numero antes de la funcion void square(*int)
  es:"<<*ptr_num<<endl;
  square(ptr_num);
  cout<<"El valor de la variable numero despues de la funcion void square(*int)
  es:"<<*ptr_num<<endl;
}
```

Responda:

1. Copie el programa anterior, compílelo y ejecútelo
2. ¿Cuál es la diferencia entre paso por valor, paso por referencia y paso por puntero?. ¿Cuál es la diferencia entre C y C++ con respecto al tipo de paso de parámetros?
3. En el programa anterior hay dos funciones que se llaman igual: ¿Cómo hace el compilador para saber cual usar?

Punteros vs Arreglos

```
#include <iostream>
using namespace std;

void cambiar_arreglo(int mi_arreglo[]){
  for(int i=0; i<10; i++){
    //Estas dos l\'ineas realizan la misma funci\'on
    //mi_arreglo[i] = mi_arreglo[i]*10;
    mi_arreglo[i] *= 10;
  }
}

int main(void){
  int *ptr_arreglo, numero=10, *ptr_num;
  int arreglo[]={9,8,7,6,5,4,3,2,1,0};
  //Arreglos vs punteros
  cout << "\n\n\nEl arreglo contiene los datos"<<endl;
  for(int i=0; i<10; i++){
    cout<<i<<") "<<arreglo[i]<<endl;
  }
```

```
  cout << "\n\n\nEl arreglo, despues de la funcion cambiar_arreglo, contiene los
  datos"<<endl;
  cambiar_arreglo(arreglo);
  for(int i=0; i<10; i++){
    cout<<i<<") "<<arreglo[i]<<endl;
  }
  cout<<"\n\n\n";
  //Y que tal con un puntero
  ptr_arreglo=arreglo;
  //arreglo=ptr_arreglo; //Esto se ocupar\'a m\'as adelante
  cambiar_arreglo(ptr_arreglo);
  for(int i=0; i<10; i++){
    cout<<i<<") "<<ptr_arreglo[i]<<endl;
  }
  cout<<"\n\n\n";
}
```

- Copie y compile el programa anterior.
- ¿Cuál es la diferencia entre un puntero y un arreglo, en el programa se vio que un arreglo se puede asignar a un puntero, descomente la línea que hace lo opuesto y compile de nuevo. ¿Qué sucede?

Punteros a punteros

Considere el siguiente programa.

```
#include <iostream>
using namespace std;
#define SIZE 25

void dar_memoria_puntero(int *mi_arreglo){
  mi_arreglo=new int[SIZE];
  for(int i=0; i<SIZE; i++){
    mi_arreglo[i]=i;
  }
  cout<<"Segun la funcion dar_memoria_puntero el arreglo contiene:"<<endl;
  for(int i=0; i<SIZE; i++){
    cout<<i<<") "<<mi_arreglo[i]<<endl;
  }
}

void dar_memoria_doble_puntero(int **mi_arreglo){
  (*mi_arreglo)=new int[SIZE];
  for(int i=0; i<SIZE; i++){
    (*mi_arreglo)[i]=i;
  }
  cout<<"Segun la funcion dar_memoria_doble_puntero el arreglo
  contiene"<<endl;
  for(int i=0; i<SIZE; i++){
```

```
    cout<<i<<") "<<(*mi_arreglo)[i]<<endl;
  }
}

int main(void){
  int *ptr_arreglo;
  //Prueba de necesidad de dobles punteros
  dar_memoria_puntero(ptr_arreglo);
  cout<<"\n\n\nDespues de la funcion dar_memoria_puntero el arreglo
  contiene"<<endl;
  for(int i=0; i<SIZE; i++){
    cout<<i<<") "<<ptr_arreglo[i]<<endl;
  }
  dar_memoria_doble_puntero(&ptr_arreglo);
  cout<<"\nDespues de la funcion dar_memoria_doble_puntero el arreglo
  contiene"<<endl;
  for(int i=0; i<SIZE; i++){
    cout<<i<<") "<<ptr_arreglo[i]<<endl;
  }
}
```

Responda:

1. Compile y pruebe el programa anterior.
2. ¿Porqué los cambios hechos a un puntero simple, no se mantienen, al regresar la función?
3. Haga un diagrama, de la localización de los punteros al ejecutarse la función, sea en el área de variables, heap o pila

Punteros a Funciones

```
#include <iostream>
using namespace std;
#define SIZE 25

int square(int mi_numero){
  return(mi_numero*mi_numero);
}

int cube(int mi_numero){
  return(mi_numero*mi_numero*mi_numero);
}

int power(int mi_numero, int(*calc_power)(int)){
  return(calc_power(mi_numero));
}

int main(void){
```

```
    //Prueba de punteros a funciones.
    cout<<"\n\nSe pasa una funcion como parametro"<<endl;
    cout<<"Se pasa la funcion square "<<power(SIZE,square)<<endl;
    cout<<"Se pasa la funcion cube "<<power(SIZE,cube)<<endl;
}
```

Responda:

1. Compile y pruebe el programa anterior.
2. ¿Qué hace la directiva *define SIZE 25*?. ¿Qué ventajes y desventajas tiene el uso de macros? ¿Cómo se obtiene un resultado equivalente, pero esta vez mediante C++?
3. Realice los cambios necesarios para poder realizar las operaciones anteriores con tipos de datos: float y double.
4. Menciones al menos 3 utilidades de punteros a funciones
5. Actualice los repositorios con los nuevos archivos.