Universidad de Costa Rica
Facultad de Ingeniería
Escuela de Ingeniería Eléctrica
Introducción a la Computación (IE-0217)
Prof: Roberto Rodríguez Rodríguez

Laboratorio #1: Recordando C y un poco de C++

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 .h y por el archivo compilado de la implementación de la librería.

El archivo .h contiene la definición de funciones, clases, métodos que se han implementado en la librería, esto es lo único que necesita conocer el programador, que tiene la librería no como se hizo.

Para incluir una librería se hace uso de la directiva #include"mi_librería" o #include<librería_del_sistema>, 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 "".

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;
```

Hola Mundo!

Este es el primer programa que típicamente se hace al aprender un lenguaje de programación. El siguiente código muestra el programa Hola Mundo en C++:

```
#include<iostream>

int main(void)
{
   std::cout<<"Hola Mundo!" <<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 un error.

Lo siguiente es la librería que se incluye, la librería *iostream* contiene definiciones de clases que sirven para comunicarse con el *standar input,* y con el *standar output*, también contiene métodos para darle formato a los datos.

El objeto *cout* sirve para enviarle cosas al *standar 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 Hola Mundo!. Al compilar obtuvo un error, esto se debe a que endl también está en el espacio de nombres estándar.

```
#include<iostream>

using namespace std;

int main(void)
{
   cout<<"Hola Mundo!" <<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 hola.cpp, para crear un programa ejecutable se debe de ejecutar el siguiente comando:

```
g++ -o hola hola.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, p.e .exe,

sólo se le debe poner la bandera de ejecución.

Para ejecutar el programa sólo se debe ejecutar esto

./hola

Por qué se llama el programa de esta manera y no simplemente digitando hola?

1. Copie el programa Hola Mundo!
2. Cree el ejecutable hola.
3. Corra el programa hola.

Funciones y Compilación de fuentes Múltiples

Cuando se trabaja con proyector grandes lo mejor es dividir el proyecto en fuentes múltiples, en estas se implementarán clases y se tendrán funciones grandes.

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

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

hola.h

```
bool hola(void);
```

hola.cpp

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

bool hola(void)
{
   cout<<"Hola Mundo!" <<endl;
   return true;
}
```

Por último se tendrá un programa principal que hará uso de la función hola, este será principal.cpp.

principal.cpp

```
#include "holaf.h"
int main(void)
{
   hola();
   return 0;
}
```

Cuando se tienen múltiples fuentes, se deben compilar primero todas las fuentes que utiliza la función main, y después crear el archivo ejecutable.

Para compilar una fuente se hace llamando al compilador con la bandera c, está 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 holaf.cpp se ejecuta:

g++ -c holaf.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 holaMF principal.cpp hola.o

Con este comando se le dice al compilador, cree el archivo ejecutable holaMF, tomando la función main de principal.cpp y uniéndolo a hola.o. Al hacer esto el compilador compila principal.cpp y se "convierte" en un linker el cual une todas las dependencias que existen, y crea el ejecutable.

4. Cree los archivos holaf.h, holaf.cpp, principal.cpp
5. Compile holaf.cpp y asegúrese que se creo el archivo hola.o, ábralo y observe que hay adentro.
6. Cree el ejecutable holaMF
7. Ejecute holaMF

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 o horas (recuerde cuando compilo el kernel). 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.

La mejor forma para aprender a hacer *makefiles* es ver alguno hecho

El *makefile* para compilar el programa hola mundo! original sería.
#Esto es un comentario

```
#Se definen los objetos, estos son los archivos que necesitan para crear el ejecutable.
OBJS = principal.cpp
#Se define el compilador
CC = g++
#Bandera de depuración
DEBUG = -g
#Banderas de compilación
CFLAGS = -Wall -c $(DEBUG) -pedantic
#Banderas para el Linker
LFLAGS = -Wall $(DEBUG) -pedantic
#Archivo Ejecutable que se va a crear
TARGET = principal

#Lo que este después de all hasta la próxima etiqueta serán las reglas a ejecutar al teclear 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 Hola Mundo! con mútiples fuentes el *makefile* sería:

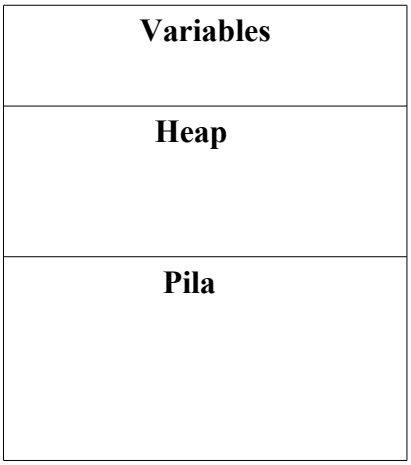
```
OBJS = principal.cpp holaMF.o
CC = g++
DEBUG = -g
CFLAGS = -Wall $(DEBUG) --pedantic -c
LFLAGS = -Wall --pedantic $(DEBUG)
TARGET = principal

$(TARGET) : $(OBJS)
    $(CC) $(LFLAGS) $(OBJS) -o $(TARGET)

holaMF.o : holaMF.h holaMF.cpp
    $(CC) $(CFLAGS) hola.cpp

clean:
        \rm -f  *.o  $(TARGET)
```

8. 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.
9. Compruebe que los programs generados funcionan.

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:



Todas las variables que se crean el área de memoria reservado para esto, asimismo, ahí se crean los punteros, un puntero puede señalar una posición de memoria de cualquiera de los tres bloques. El área *heap* se utiliza para la asignación de memoria dinámica, (esto se verá en otra clase), 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 y estos mismos.

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

//*************************************************************************
void calc_cuadrado(int *ptr_mi_numero)
{
   *ptr_mi_numero=*ptr_mi_numero**ptr_mi_numero;
   cout<<"Segun la funcion void calc_cuadrado(int *ptr_mi_numero) el valor de numero es:
"<<*ptr_mi_numero<<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 cuadrado(int) es: "<<numero<<endl;
   calc_cuadrado(numero);

   cout<<"El valor de la variable numero despues de la funcion void cuadrado(int) es:
"<<numero<<endl;

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

   cout<<"El valor de la variable numero despues de la funcion void cuadrado(*int)
es:"<<numero<<endl;

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

   calc_cuadrado(ptr_num);

   cout<<"El valor de la variable numero despues de la funcion void cuadrado(*int)
es:"<<*ptr_num<<endl;
}
```

10.Copie el programa anterior, compílelo y ejecútelo

En el programa anterior hay dos funciones que se llaman igual, Cómo hace el compilador para saber cual usar? Eso se llama homonimía o sobrecarga de funciones.

Punteros vs Arreglos

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

//*************************************************************************
void cambiar_arreglo(int mi_arreglo[])
{
   for(int i=0; i<10; i++)
   {
      //mi_arreglo[i]=mi_arreglo[i]*10; //Estas dos líneas realizan la misma función
      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 ocupará más adelante
    cambiar_arreglo(ptr_arreglo);
    for(int i=0; i<10; i++)
    {
       cout<<i<<"-) "<<ptr_arreglo[i]<<endl;
    }
    cout<<"\n\n\n";
}
```

11.Copie y compile el programa anterior.
12.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;
   }
}
```

13. Compile pruebe el programa anterior.
14 .Por qué los cambios hechos a un puntero simple, no se mantienen, al regresar la función?
15. 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 cuadrado(int mi_numero)
{
   return(mi_numero*mi_numero);
}
//*************************************************************************

//*************************************************************************
int cubo(int mi_numero)
{
   return(mi_numero*mi_numero*mi_numero);
}
//*************************************************************************

//*************************************************************************
int potencia(int mi_numero, int(*calc_potencia)(int))
{
   return(calc_potencia(mi_numero));
}
//*************************************************************************

int main(void)
{
   //Prueba de punteros a funciones.
   cout<<"\n\nSe pasa una funcion como parametro"<<endl;
   cout<<"Se pasa la funcion cuadrado "<<potencia(SIZE,cuadrado)<<endl;
   cout<<"Se pasa la funcion cubo "<<potencia(SIZE,cubo)<<endl;
}
```

16. Compile y pruebe el programa anterior.
17.Piense en utilidades de punteros a funciones