LABORATORIO I
ICOM 4015 ADVANCED PROGRAMMING

1. Objetivos.

a. Entender como se construyen programas en forma modular. Uso de funciones.
b. Cual es el propósito de las reglas de alcance.

Las funciones son herramientas poderosas. Son segmentos de código las cuales pueden ser invocadas por otras partes del programa. Esto permite al código ser escrito con *reusabilidad*. Una función podría ser razonablemente general y contener cierta parte de código. Si una parte de código es usado distintas veces en diferentes puntos en un programa es buena idea colocar esto dentro de una función. Esto permite al programador modularizar un programa y obtener un programa más optimizado.

El alcance de una variable, es la región de código en la cual una declaración esta activa. Por ejemplo, cuando se declara una variable como local en un bloque, esta solo puede ser referenciada en ese mismo bloque, o en bloques internos a este. Es una buena práctica limitar el alcance de las variables lo más posible para dar mantenimiento al código.

El propósito principal de las reglas de alcance es permitir el desarrollo de secciones independientes de código de tal manera que aún cuando si los identificadores idénticos son usados para referenciar a diferentes objetos no exista un conflicto, y también para permitir al programa para re-usar la memoria previamente asignada a diferentes variables.

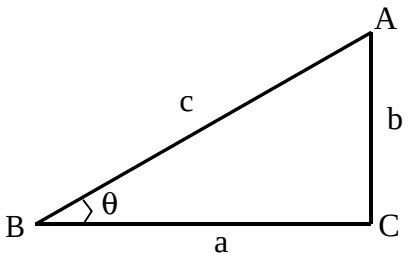
La declaración de variables estáticas (static) tienen un efecto diferente dependiendo de si la variable es declarada globalmente o a nivel de función. Cuando las variables estáticas son declaradas en un alcance global, estas sólo pueden ser accesadas por funciones definidas dentro de este archivo fuente. Pero si las variables son declaradas como estáticas dentro de la definición de una función, a estas se les asignan unos espacios de almacenamiento y son inicializadas antes de que comience la ejecución del programa. Estas permanecen hasta que el programa finaliza.

2. Ejemplos

a. Función para calcular el factorial de un número. Este código calcula el valor de n! o también conocida como Operación Factorial.

```
int factorial(int n)
{
  int counter;
  int x;
  counter = 1;
  x = 1;
  while (counter <= n)  {
       x *= counter;
       counter++;
   }
   return x;
}
```

b. Mas funciones. Uso de funciones trigonométricas.



$$\text{Sin } \theta = \frac{b}{c} \qquad \text{Cos } \theta = \frac{a}{c} \qquad c^2 = a^2 + b^2$$

Pero también es posible obtener un estimado del valor de la función trigonométrica, utilizando la aproximación de MacLaurin.

$$\sin(x) = \sum (-1)^n \bullet \frac{x^{2n+1}}{(2n+1)!} \qquad \cos(x) = \sum (-1)^n \bullet \frac{x^{2n}}{(2n)!}$$

```
extern "C"{
#include <math.h>
}
#include <iostream>

double factorial(double n);
double sine(double x, double numAprox);
double cosine(double x, double numAprox);

int main(void) {
  double aprox, angle;
  cout << "Valor del angulo (en radianes) ";
  cin >> angle;
  cout << "Numero de terminos para hacer la aproximacion ";
  cin >> aprox;
  cout << "sin(" << angle << ") = "
       << sine(angle, aprox) << endl;
  cout << "cos(" << angle << ") = "
       << cosine(angle, aprox) << endl;
}

double factorial(double n) {
  int counter;
  double x;
  counter = 1;
  x = 1;
  while (counter <= n)  {
    x *= counter;
    counter++;
  }
  return x;
}

double cosine(double x, double numAprox) {
```

```
  double cos = 0;
  for (double i=0;i<=numAprox+1;i++)
    cos += ((pow(-1, i)) * (pow(x, 2*i))) / (factorial(2*i));

  return cos;
 }

 double sine(double x, double numAprox) {
  double sin = 0;
  for (double i=0;i<=numAprox+1;i++)
    sin += ((pow(-1, i)) * (pow(x, (2*i)+1))) / (factorial((2*i)+1));

  return sin;
 }
```

c. Alcance de una variable.

```
#include <iostream.h>

void Increment(void);
void Print();

 int global;

int main(void) {
   int i;
   for (i = 0; i != 10; i ++)
     Increment();
   Print();
   return 0;
}

void Increment(void) {
    static int number = 100;
    / / number es iniciada a 100 sólo en el primer llamado a la función Increment().

    cout << number++ << endl;

    / / en cada llamado el valor de number es de salida.

    cout << global << endl;

    / / en cada llamado el valor de global es el mismo, puesto que no se modifica.
}

void Print() {
   cout << global << endl;
    / / global puede ser accesada por cualquiera de las funciones del programa.
}
```

3. Ejemplo de situación repetitiva

Menú 1 Menú principal
1. Proceso de Estudiantes
2. Proceso de Profesores
3. Proceso de Empleados
4. Salir del programa

Menú 1.1 Proceso de Estudiantes
1. Entrar datos
2. Modificar datos
3. Borrar datos
4. Salir a menú anterior
5. Salir del programa

Menú 1.1.1 Entrar datos
1. Datos personales
2. Datos académicos
3. Salir a menú anterior
4. Salir del programa

Menú 1.1.2 Modificar datos
1. Datos personales
2. Datos académicos
3. Salir a menú anterior
4. Salir del programa

Menú 1.1.3 Borrar datos
1. Datos personales
2. Datos académicos
3. Salir a menú anterior
4. Salir del programa

Menú 1.2 Proceso de Profesores
1. Entrar datos
2. Modificar datos
3. Borrar datos
4. Salir a menú anterior
5. Salir del programa

Menú 1.2.1 Entrar datos
1. Datos personales
2. Datos académicos
3. Salir a menú anterior
4. Salir del programa

Menú 1.2.2 Modificar datos
1. Datos personales
2. Datos académicos
3. Salir a menú anterior
4. Salir del programa

Menú 1.2.3 Borrar datos
1. Datos personales
2. Datos académicos
3. Salir a menú anterior
4. Salir del programa

Menú 1.3 Proceso de Empleados
1. Entrar datos
2. Modificar datos
3. Borrar datos
4. Salir a menú anterior
5. Salir del programa

Menú 1.3.1 Entrar datos
1. Datos personales
2. Datos académicos
3. Salir a menú anterior
4. Salir del programa

Menú 1.3.2 Modificar datos
1. Datos personales
2. Datos académicos
3. Salir a menú anterior
4. Salir del programa

Menú 1.3.3 Borrar datos
1. Datos personales
2. Datos académicos
3. Salir a menú anterior
4. Salir del programa

a. Posible solución: (Usando arreglos y archivos se puede conseguir una solución más general)

```
#include <iostream.h>

void menu(char *title; int t, m1, m2, m3, m4, m5) {
 int mess = 10;

 cout << "Menu " << title;
 if (t == 1) cout << " Menu principal" << endl;
 else if (t == 2) cout << " Proceso de Estudiantes" << endl;
 else if (t == 3) cout << " Proceso de Profesores" << endl;
 else if (t == 4) cout << " Proceso de Empleados" << endl;
 else if (t == 5) cout << " Entrar datos" << endl;
 else if (t == 6) cout << " Modificar datos" << endl;
 else if (t == 7) cout << " Borrar datos" << endl;
```

```
  if (m1 == 1) cout << "1. " << " Proceso de Estudiantes" << endl;
  else if (m1 == 2) cout << "1. " << "Entrar datos" << endl;
  else if (m1 == 3) cout << "1. " << "Datos personales" << endl;

  if (m2 == 1) cout << "2. " << " Proceso de Profesores" << endl;
  else if (m2 == 2) cout << "2. " << "Modificar datos" << endl;
  else if (m2 == 3) cout << "2. " << "Datos academicos" << endl;

  if (m3 == 1) cout << "3. " << " Proceso de Empleados" << endl;
  else if (m3 == 2) cout << "3. " << "Borrar datos" << endl;
  else if (m3 == 3) cout << "3. " << "Salir a menu anterior" << endl;

  if (m4 == 1) cout << "4. " << " Salir del programa" << endl;
  else if (m4 == 2) cout << "4. " << "Salir a menu anterior" << endl;

  if (m5 == 1) cout << "5. " << " Salir del programa" << endl;
}
```

4. Asignación I: Series de Fibonacci.

Realice el programa en c++ (utilizando funciones) que cree una serie de Fibonacci, dado que dicha series es:

0, 1, 1, 2, 3, 5, 5, 8, 13, 21, ...

y empieza con los elementos 0 y 1, y de ahí en adelante, cada nuevo elemento es la suma de los dos anteriores. La definición de la serie de Fibonacci es la siguiente:

fibonacci(0) = 0
fibonacci(1) = 1
fibonacci(n) = fibonacci(n - 1) + fibonacci(n - 2)