LABORATORIO III
ICOM 4015 ADVANCED PROGRAMMING

1. Objetivos.

a. Repaso de los conceptos vistos hasta el momento.
b. Overloading y templates.

Haremos un breve repaso de los temas tratados en la clase, con miras a la presentación del primer examen parcial del curso. Los temas son:

- **I/O streams (basics)**

```
#include <iostream.h>
#include <iomanip.h>

int main()
{
  cout << setw(3) << 6 << endl
       << setw(3) << 18 << endl
       << setw(3) << 124 << endl
       << "---\n"
       << (6+18+124) << endl;

  return 0;
}
```

- **Abstracción de procedimientos**
  - ¤ Funciones como contratos abstractos
  - ¤ El qué *vs.* el cómo

  La asignación de un nombre a un tipo de datos, se conoce como *data abstraction*. La asignación de un nombre a una función o procedimiento, para que cada vez que se necesite invocar a dicha función, solo sea necesario usar el nombre de la función, seguido de los argumentos apropiados, esto se conoce como *procedural abstraction*.

```
int leapyear(int year)
{
  // Preconditions: the parameter year must represent a year in a four
  //              : digit form, such as 1999

  // Postcondition: a 1 is returned if the year is a leap year;
  //             : otherwise a 0 will be returned

  // leap year occurs every fourr years, excepts for years ending in 00, in
     which case only it the year is divisible by 400.

  if ( (year % 400 == 0 && year % 100 ¡= 0) || (year % 400 == 0))
    return 1;       // is a leap year
  else
    return 0;       // is not a leap year
}
```

- **Mecanismos de paso de parámetros**
  - ¤ Por valor
  - ¤ Por referencia

El uso de los parámetros por referencia permite que una función cambie el valor del parámetro actual o argumento, el cual es usado cuando se llama la función. Por defecto, el método utilizado es el de pasar argumentos por valor, lo cual implica que la función invocada no puede cambiar el parámetro real. Esto permite tener una cierta protección de los datos originales. Utilizamos solamente variables por referencia cuando deseamos cambiar el valor de un parámetro real y además permite ahorrar un poco de memoria en el proceso.

```
#include <iostream.h>

// Forward definitions
void swap(float& num1, flota& num2);

int main() {
  float firstNum = 20.5, secNum = 6.25;

  cout << "The value stored in firstNum is: " << firstNum << endl;
  cout << "The value stored in secNum is: " << secNum << endl;

  swap(firstNum, secNum);   // call the function with references

  cout << "The value stored in firstNum is now: " << firstNum << endl;
  cout << "The value stored in secNum is now: " << secNum << endl;

  return 0;
}

void swap(float& num1, flota& num2);
  float temp;

  temp = num1;
  num1 = num2;
  num2 = temp;

  return;
}
```

- **Parámetros funcionales**

```
#include <iostream.h>

// Forward definitions
double f(double x);
void plot(double fcn(double), double x0, double increment, int limit);

int main() {
  cout << "Mapping function (x^2) + (1/x) " << endl;
  plot(f, 0.01, 0.01, 100);
}


double f(double x) {
  return (x * x + 1 / x);
}
```

```
void plot(double fcn(double), double x0, double increment, int limit) {
  for (int i = 0; i<limit;i++) {
    cout << " x : " << x0
         << "   f(x) : " << fcn(x0) << endl;
    x0 += increment;
}
```

- **Scoping**

  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.

- **Top-down design stepwise refinement**

- **Algoritmos de integración y diferenciación**

  Ver ejemplos en el website del curso.

- **Números pseudo-aleatorios**

```
#include <iostream.h>
#include <iomanip.h>
#include <stdlib.h>
#include <time.h>

const int NUMBERS = 10;

// This program generates ten pseudorandom numbers
// using C++'s rand() function

int main() {
  float randvalue;
  for (int i=1; i<= NUMBERS;i++) {
     randvalue = rand();
     cout << setw(20 << randvalue << endl;
  }

  return 0;
}
```

- **Recursion vs. iteración**
  - Activation records
  - Call stacks
  - Duplicación de trabajo

- **Análisis de algoritmos**
  - Tiempo lineal *vs.* tiempo exponencial

- **Overloading y Templates**

  C++ utiliza *template* para proporcionar polimorfismo paramétrico, lo cual permite que el mismo código sea utilizado con respecto a varios tipos de variables, donde el tipo es un parámetro en el código. Ésta es una forma de programación genérica. Usar *templates* para definir clases y funciones permite que reutilicemos código de una manera simple, segura y que permite al compilador automatizar el proceso del instanciación de las variables.

  Las funciones con *overloading* son una adición importante en C++. *Overloading* significa que es seleccionado dependiendo de los argumentos de la función. Cuando se invoca una función con *overloading*, el compilador debe tener un algoritmo de selección con el cual, escoge la función apropiada. El algoritmo que logra esto depende de qué tipo de conversiones están disponibles.

  Ejemplos:

```
template <class TYPE>
void copy(TYPE a[], TYPE b[], int limit) {
  for (int i = 0; i<limit;i++) {
     a[i] = b[i];
}
```

  La invocación de la función *copy()* con sus argumentos específicos, causa que el compilador genera una función basada en esos argumentos. Si no es posible, se generara un error en tiempo de compilación. Por ejemplo, si definimos las siguientes variables:

```
double f1[50], f2[50];
char c1[25], c2[25];
int i1[75], i2[75];
```

  Si utilizamos la función *copy()* con dichas variables, podríamos escribir algo como:

```
copy (f1, f2, 50);
copy (c1, c2, 10);
copy (i1, i2, 40);
copy (i1, f2, 50);
```

  La ultima invocación de *copy()* fallan al compilar porque los tipos de los argumentos no pueden ser encontrados en la definición de *template*.