UCAB UNIVERSIDAD CATÓLICA ANDRÉS BELLO

Algoritmos y Estructuras de Datos

Oscar Meza

omezahou@ucab.edu.ve

PRESENTACIÓN

Oscar Meza

- Licenciado en Computación *Universidad Central de Venezuela*
- Doctor en Investigación de Operaciones *Université Joseph Fourier, Grenoble, Francia*
- Profesor Titular jubilado de la *Universidad Simón Bolívar, Caracas, Venezuela*
- Profesor colaborador de la VIU, España
- Especializado en teoría y aplicaciones de Grafos.
- Mi investigación ha sido sobre todo en *Problemas de Optimización Combinatoria*, más concretamente en *Problemas de Enrutamiento en Grafos*

Esta asignatura tiene como objetivo presentar los temas

- Fundamentos del lenguaje de programación C++ el cual utilizaremos a lo largo del curso para desarrollar nuestros programas.

- Diseño e implementación de los tipos de datos abstractos (TDA) básicos en computación (**listas, pilas, colas, colas con prioridades, árboles, tablas de hash y grafos**) que permiten almacenar de manera estructurada los datos que utilizamos en nuestros programas. Para ello utilizaremos manejo dinámico de la memoria (**apuntadores o punteros**).

Unidad de Competencia 1: Lenguajes de programación estructurada para la gestión dinámica de memoria.

1. Tipos de lenguajes de programación.
2. Aplicaciones.
3. Compiladores.
4. Intérpretes.
5. Estructura de un programa, tipos de datos, comentarios, operadores, estructuras selectivas, estructuras iterativas.
6. Funciones.
7. Uso de librerías.

Unidad de Competencia 2: Punteros y gestión de memoria.

1. Gestión de memoria dinámica: direcciones de memoria y apuntadores.
2. Definición de apuntadores y su utilidad.
3. Operadores: Dirección y contenido.
4. Operaciones básicas con apuntadores.
5. Apuntadores y arreglos.
6. Gestión de memoria dinámica.
7. El operador new.
8. Arreglos de datos y arreglos dinámicos.

Unidad de Competencia 3:

Estructuras de datos dinámicas secuenciales: Listas.

1. Representaciones estáticas Vs. Representaciones dinámicas.

2. TDA Lista: simples, dobles, circulares y multi-enlazadas.

3. TDA: Pila.

4. TDA: Cola.

Unidad de Competencia 4:

Estructuras de datos dinámicas Jerárquicas: Árboles.

1. Árboles Generales. Árboles Binarios.
2. Algoritmos de recorrido de árboles: preorden, inorden, postorden.
3. Árboles Binarios de Búsqueda.
4. Árboles AVL.
5. Árboles B-TREE.
6. Árboles Rojo-negro.
7. Algoritmos de ordenamiento basados en Árboles (Montículos o Heaps): Heapsort.

Unidad de Competencia 5:

Estructuras dinámicas jerárquicas: Grafos.

1. Grafos y sus aplicaciones.
2. Representación de grafos: Matriz de adyacencia y matriz de incidencia, listas de adyacencias.
3. Recorridos de Grafos: Algoritmos DFS, Algoritmo BFS, Algoritmo de Dijkstra, Algoritmo Roy Warshall.
4. Algoritmos de recubrimiento mínimo: algoritmo de Kruskall y algoritmo de Prim.
5. Coloración de grafos: número cromático, algoritmo de Greedy y algoritmo de Brelaz.

Al finalizar esta asignatura se espera que el estudiante sea capaz de:

- Conocer, Diseñar, implementar y evaluar las implementaciones de los tipos de datos abstractos fundamentales LISTA, PILA, COLA, COLA CON PRIORIDADES, TABLA DE HASH, ÁRBOLES y GRAFOS.
- Utilizar el lenguaje de programación orientado a objetos C++ (sus fundamentos).

Libros de texto:

- Benjumea, Vicente y Roldán, Manuel (2022). ***Fundamentos de Programación con el Lenguaje de Programación C++.*** Universidad de Málaga. (En Modulo 7)
- Luis Joyanes, Ignacio Zahonero (2004) ***Algoritmos y Estructuras de datos. Una perspectiva en C***. MacGraw-Hill). (En Modulo 7)
- Thomas H. Cormen, Charles E. Leiserson, Ronald L. Rivest, Clifford Stein (2009). ***Introduction to Algorithms, 4rd Edition*** **(The MIT Press)** https://dl.ebooksworld.ir/books/Introduction.to.Algorithms.4th.Leiserson.Stein.Rivest.Cormen.MIT.Press.9780262046305.EBooksWorld.ir.pdf

Bibliografía complementaria

- **C++ Reference** documents the C++ Standard Library (pueden buscar cualquier información de C++) https://en.cppreference.com/w/
- **Halterman Richard**. (2023). ***Fundamentals of C++ Programming*.** School of Computing Southern Adventist University https://python.cs.southern.edu/cppbook/progcpp.pdf
- **Clifford A. Shaffer (2013).Data Structures and Algorithm Analysis. Department of Computer Science Virginia Tech**. https://people.cs.vt.edu/~shaffer/Book/C++3elatest.pdf
- **Mark Allen Weiss (2003). Data Structures and Solving Problems using C++. Pearson https://cdn.preterhuman.net/texts/math/Data_Structure_And_Algorithms/**
- **Dietel (2004). Como programar en C, C++ y JAVA. Pearson. https://www.academia.edu/89227219/Como_Programar_C_C_y_Java_4ta_Edici%C3%B3n_Deitel**
- **Oscar Meza; Maruja Ortega**. "Grafos y Algoritmos". Editorial Equinoccio, Segunda edición mejorada. Universidad Simón Bolívar. 2007. ISBN 980-237232-3. Libro de texto en la USB (el PDF está en Modulo 7)

Para compilar y correr los programas en C++ lo haremos en Visual Studio Code (Vscode): un editor configurable a la mayoría de lenguajes de programación que permite ejecutar programas en C++.

Existen varios IDEs (ambientes integrados de desarrollo) para C++ que además de poseer el editor poseen mas funciones que VScode y ejecutar de manera más amigable los programas, como por ejemplo Visual Studio 2022 de Microsoft, Eclipse, etc.

Página de instalación: https://code.visualstudio.com/docs/cpp/config-mingw
Video de instalación: https://www.youtube.com/watch?v=iPvvbNVisfU

En laboratorio lo veremos y utilizaremos

Se sugiere que aprendan GIT y GITHUB (software de control de versiones) .....

Material para aprender a trabajar con GIT y GITHUB :

Presentación de GIT:

https://platzi.com/cursos/gitgithub/

Cursos desde CERO:

https://youtu.be/3GymExBkKjE?si=zK7Y6oW-CgW8hqtV

https://youtu.be/mBYSUUnMt9M?si=DGkSuh0TgVhrOqQk

Trabajar en equipo:

https://youtu.be/g-coH1XtqrQ?si=HtdBvqyn3bSCJ3DE

- **La presentación PDF de cada clase estará accesible en MÓDULO 7**

- **EVALUACIÓN:**
- **2 exámenes parciales (20% cada uno de la nota final)**
- **2 quices (5% cada uno de la nota final)**
- **2 talleres (5% cada uno de la nota final)**
- **2 proyectos (20% cada uno de la nota final)**

- **LA PRIMERA SEMANA DE CLASES FORMAR GRUPOS DE TRES PERSONAS para realizar los proyectos (enviarme email a omezahou@ucab.edu.ve con los integrantes del grupo)**

- **Fechas de las evaluaciones están en el programa del curso en Módulo 7:**
  - **Primer parcial (2 horas): 19 de mayo**
  - **Segundo parcial (2 horas): 7 de julio**
  - **Quiz 1 (1/2 hora): 25 de abril**
  - **Quiz 2 (1/2 hora): 9 de junio**
  - **Proyecto 1: se publica 31 de marzo y se entrega el 12 de mayo, evaluación individual el 19 de mayo**
  - **Proyecto 2: se publica 16 de mayo y se entrega 23 de junio, evaluación individual el 30 de junio**
  - **Taller 1: 12 de mayo**
  - **Taller 2: 23 de junio**

- Para aprobar la asignatura deberá:
  - Tener al menos 4.75 puntos en la suma de las notas de los exámenes y quices,
  - Tener al menos 4.75 en la suma de las notas de los proyectos y los talleres.

Dudas y preguntas

- Las consultas las puede hacer al final de clases.

- Por email: omezahou@ucab.edu.ve

Lenguajes de programación imperativa (programación estructurada):

- Son lenguajes donde un programa es una secuencia de instrucciones que se ejecutarán siguiendo la secuencia.
- Las instrucciones básicas son la asignación, instrucciones condicionales (if..) e instrucciones iterativas (while, for..)
- Como ejemplos están: JAVA, PYTHON, C++. Aunque estos tres lenguajes permiten también programar mediante el paradigma ORIENTADO POR OBJETOS.

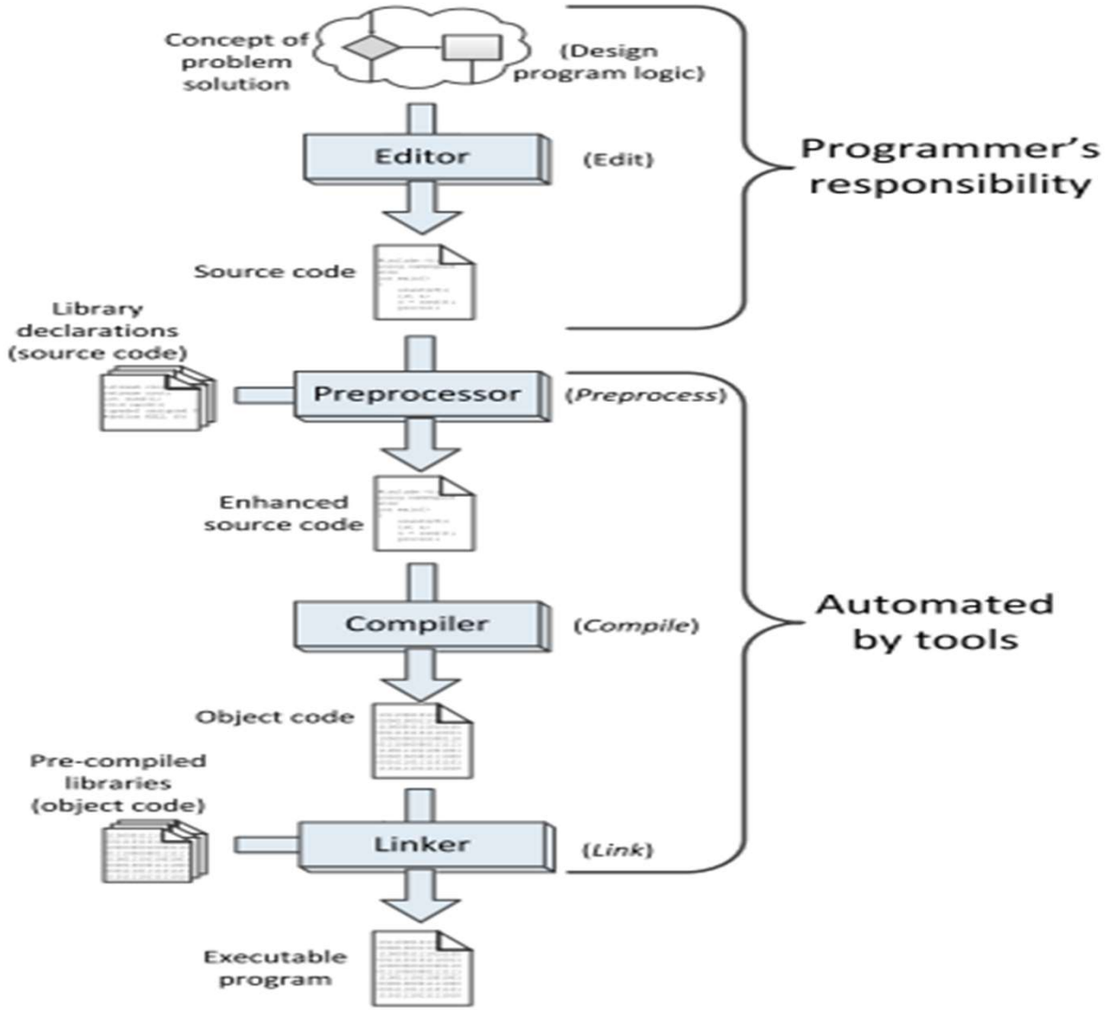
Algunos de estos lenguajes son compilados y/o interpretados ¿qué significa?

Lenguajes compilados

Los lenguajes compilados son convertidos por un compilador, directamente a **código máquina** que el procesador puede ejecutar. Como resultado, **suelen ser más rápidos** y más eficientes al ejecutarse en comparación con los lenguajes interpretados.

Es el caso de C, C++

- Un programa en C++ para ser ejecutado, require ser traducido por un compilador a un programa ejecutable en código de máquina (a diferencia de Python y Java que son interpretados)



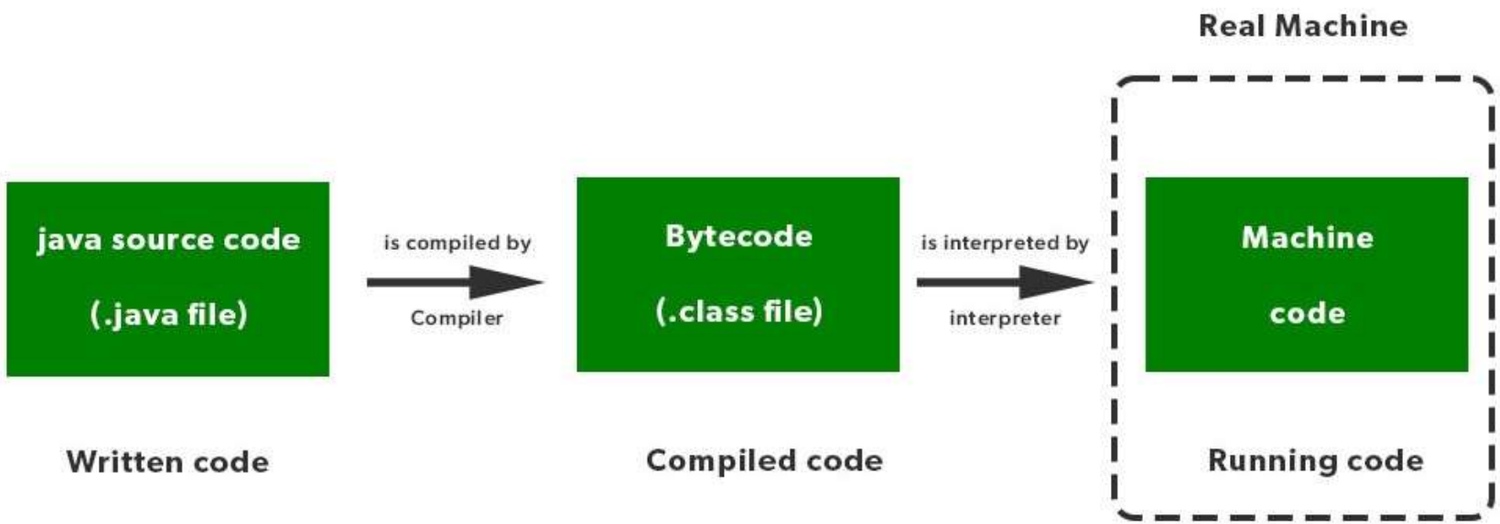
Lenguajes interpretados

Es aquel donde el **código fuente** se ejecuta directamente, instrucción a instrucción mediante un **intéprete**.
Es decir, **el código no pasa por un proceso de compilación**, sino que **tenemos** un programa llamado **intérprete** que lee la instrucción en tiempo real, y la ejecuta.

Es el caso de PYTHON, JAVA.

Sin embargo JAVA es “semi-compilado”

Primero se compila a un formato de archivo llamado **BYTECODE** y es este archivo que es ejecutado por el intérprete.



La mayoría de lenguajes de programación pueden tener implementaciones tanto compiladas como interpretadas**, el lenguaje en sí mismo no es necesariamente compilado o interpretado**. Sin embargo, por cuestiones de simplicidad, normalmente se les conoce de tal manera.

Un programa en Python, por ejemplo, puede ser ejecutado ya sea como programa compilado o interpretado en modo interactivo.

Por ejemplo, la mayoría de las herramientas de **línea de comandos (shells)** pueden ser consideradas como lenguajes interpretados.

Ventajas y desventajas de los lenguajes compilados

Ventajas de los lenguajes compilados:

Los programas que son compilados a un código máquina nativo **suelen ser más rápidos** que los lenguajes interpretados pues se ajustan al hardware.

Desventajas de los lenguajes compilados:

Dependencia de la plataforma del código binario generado (código objeto).

Ventajas y desventajas de los interpretados

Ventajas de los lenguajes interpretados:

Ya que los intérpretes ejecutan el código fuente del programa, **el código en sí es independiente de la plataforma.**

Desventajas de los lenguajes interpretados:

La desventaja más notable es la velocidad de ejecución comparada con los lenguajes compilados.

EL LENGUAJE DE PROGRAMACIÓN C++

EL LENGUAJE DE PROGRAMACIÓN C++

A diferencia de PYTHON,
C++ ES FUERTEMENTE TIPEADO (**strongly typed language**):

a toda variable hay que definirle explícitamente un tipo (integer, double, arreglo, etc.)

Ejemplo: int x; declaramos la variable x tipo entero

C++, como cualquier lenguaje de programación imperativa, consta de:

- **tipos primitivos** que pueden tener las variables (**entero, booleano, etc.)**
- **operadores** sobre las variables de acuerdo a su tipo
- **las instrucciones condicionales y de repetición (bucles)**
- **Rutinas: Funciones y procedimientos**.

C++ permite implementar el paradigma orientado a objetos

Podemos construir otros **tipos** de datos y se definen utilizando el constructor (o "constructo" del lenguaje, "construct" en inglés):

- **class** para definir un objeto y las operaciones que podemos realizar sobre ese tipo de objeto (así como un entero es un tipo de objeto que posee las operaciones suma, resta, etc.)

C++ ofrece un conjunto de funciones predefinidas en el lenguaje (en su biblioteca standard), como funciones matemáticas, manejo de archivos, etc.

Sin embargo en este curso no trabajaremos con el paradigma Orientado a Objetos que ofrece C++: solo utilizaremos objetos ya definidos (el tipo **vector** por ejemplo)

Sin embargo vamos a poder construir otros "**tipos** de datos" y se definen utilizando el constructor:

- **struct** para agrupar un conjunto de objetos, por ejemplo las coordenadas de un punto en el plano cartesiano, que están compuestas de **dos** números reales

Un primer programa en C++

Informa al compilador que utilizará la biblioteca estándar "iostream" que contiene el objeto "cout"
std:: es "demarcador de alcance" para indicar que cout está dentro de un "namespace" llamado "std"….. ¿¿…??. Luego nos familiarizaremos con esto…

```
#include <iostream>

int main() {
    std::cout << "This is a simple C++ program!\n";
}
```

Cadena de caracteres (string en inglés)

- **Todo programa en C++** debe poseer la función **main** para poder ser ejecutado

```
#include <iostream>

int main() {
    std::cout << "This is a simple C++ program!\n";
}
```

Si guardamos el programa anterior en un archivo llamado **primero.cpp** podemos abrir una ventana de comandos y ejecutar:

Importante la extensión cpp

>c++ primero.cpp -o primero

Se generará un archivo **primero.exe**, que podemos ejecutar en una ventana de comandos:

>primero ..... Tecleando “enter”
> This is a simple C++ program! ..... Este es el resultado
>

Otro programa en C++

```
#include <iostream>
Int main{
    std::cout << "    * \n" << "  *** \n" << "***** \n" << "   * \n" << "   * \n"
              << "   * \n";
}
```

Al ejecutarlo imprime:

Una versión mas estética:

```
#include <iostream>

int main() {
    std::cout << "     *     \n"
              << "    ***    \n"
              << "   *****   \n"
              << "     *     \n"
              << "     *     \n"
              << "     *     \n";
}
```

Al ejecutarlo imprime:

Tipos primitivos en C++: enteros (int, short int, long int, long long int), reales (float, double, long double), booleano (bool), Caracter (char)

Ejemplos: veamos en detalle estos dos programas

```
#include <iostream>
 int main() {
     int x;  // declaracion mas no inicializacion
     int z, y = 20;
    const int w = 20;
     x = 10;
     z = 40;
     std::cout << x << " " << y << " " << z << " " << w<< '\n';
 }
```

Comentario de línea

```
#include <iostream>
 int main() {
     double x = 7.5;
     int  y  {4};
     bool z = true;
     z = false;
     std::cout << x << " " << y << " "
                    << z << '\n';
 }
```

Palabra reservada como main, int, etc.

Hagamos juntos:

Hacer un programa que declare dos variables x e y: entero, double. Las inicialice en 40 y 45.5 y luego la impresión sea:

El valor de x = 40 y el valor de y = 45.5

Modelo:

```
#include <iostream>
 int main() {
     double x = 7.5;
     int  y  {4};
     bool z = true;
     z = false;
     std::cout << x << " " << y << " " << z << '\n';
 }
```

Los nombres de variables comienzan por una letra o un carácter _ (underscore) y luego puede tener cualquier letra, número o underscore. Ejemplo xxx, Xxx, ____xXX111.

Distingue mayúsculas y minúsculas (case sensitive). Veamos otras características:

```
#include <iostream>

int main() {
    char _aaa1122;
    char b = 'A';
    aaa_1122 = 65;  // 65 es el código ASCII de 'A' . A un char le podemos asignar a un entero

   // notación científica: 2.3e2  es equivalente a 2.3 multiplicado por 10  elevado al cuadrado

    const double c = 2.998e8;  //notación científica

     std::cout << "Velocidad de la luz= " << c << '\n';
            std::cout << "a=" <<  aaa_1122 << "  b=" << b << '\n';
}
```

¿Qué imprime?:

Velocidad de la luz= 2.998e+8
a=A b=A

C++ permite inferencia de tipos:

```
int count = 0;
char ch = 'Z';
double limit = 100.0;
```

Es equivalente a:

```
auto count = 0;
auto ch = 'Z';
auto limit = 100.0;
```

Porque, por ejemplo, al inicializar **count** con 0, el compilador infiere que es un entero, **limit** será de tipo double por haber un punto decimal

Hagamos juntos el siguiente ejercicio:

Determine el tipo exacto de cada una de las siguientes variables:

(a) auto a = 5;

(b) auto b = false;

(c) auto c = 9.3;

(d) auto d = 5.1f; (f es por flotante)

(e) auto e = 5L; (L es por long int)

(f) auto f =‘A’ ;

Operadores aritméticos y entrada por teclado (hacer en laboratorio):

```
#include <iostream>

 int main() {
     int valor1, valor2, sum;
     std::cout << “Coloque dos valores y teclee enter: ";
     std::cin >> valor1 >> valor2;  // entrada por teclado, tecleamos por ejemplo: 2 3   enter
     sum = -(valor1 + valor2)*2;
     std::cout << "-( " << valor1 << " + " << valor2 << ") * " << 2 <<   " = " << sum << '\n';
 }
```

¿Qué imprime si leemos 2 y luego 3?:

Coloque dos valores y teclee enter: 2 3
-(2 + 3) * 2 = -10

Este programa calcula la circunferencia de un circulo (= 2*PI*r) y está errado ¿Por qué? :

```
#include <iostream>
int main() {
    double C, r;
    const double PI = 3.14159;
    // Formula del area de un circulo
   C = 2*PI*r;
   // obtener el radio
   std::cout >> "Coloque el radio: ";
   cin << r;
   std::cout << "Circumference is " << C << '\n'; // Imprimir area
}
```

Este programa calcula la circunferencia de un circulo (= 2*PI*r) y está errado ¿Por qué? :

```
#include <iostream>
int main() {
    double C, r;
    const double PI = 3.14159;
    // Formula del area de un circulo
   C = 2*PI*r;   // r no tiene ningun valor aun
   // obtener el radio
   std::cout >> "Coloque el radio: ";
   cin << r;
   std::cout << "Circumference is " << C << '\n'; // Imprimir area
}
```

Algunas observaciones (aburrido…):

- Los operadores tienen un orden de precedencia, Por ejemplo 2 + 3 * 4 , la multiplicación tiene mayor precedencia que la suma. = y – tienen igual precedencia, * y / también tienen igual precedencia. Se recomienda parentizar completamente las expresiones.
- 5 / 2 devuelve la división entera de 5 entre 2, es decir 2
- Para que lo considere un “double” habría que escribir: 5.0 / 2
- Puedo asignar un entero a un “double” y viceversa (de double a int asigna la parte entera del double si está en el rango de int).
- El operador % devuelve el resto de la división entera entre dos números enteros. Ejemplo: 5%3 es 2 , -5%3 es -2

Precedencia de operadores aritméticos, se asocian de izquierda a derecha

Arity	Operators	Associativity
Unary	+, –	
Binary	*, /, %	Left
Binary	+, –	Left
Binary	=	Right

Ejemplos:

2 + 3 * 4 corresponde a 2 + (3*4)

2 - 3 – 4 corresponde a (2-3)-4 (asocia de izquierda a derecha)

2 .0 / 4 * 5 +3 correspone a ((2/4)*5)+3

Es preferible parentizar bien las expresiones para evitar ambigüedad en una expresión aritmética o de cualquier otro tipo

¿Qué valor contendrá x en la siguiente asignación?

Double x = 5 / 10 * (19.5 – 10.5)

Imprime:
0

Porque 5/9 es cero y evalua de izquierda a derecha cuando los operadores tienen igual precedencia

Más observaciones:

- Si tenemos :
  - double d = 1.6;
  - int i = d;
  - Algunos compiladores dan una advertencia (WARNING) de que se puede perder datos
  - Para evitar la advertencia se puede hacer un “cast” que permite convertir tipos compatibles. Ejemplo: **i = static_cast<int>(d);**
  - **Pero hay que tener cuidado con los rangos de valores de los tipos int y double porque d puede tener una cantidad que esté fuera del rango de i**

Más observaciones:

- Pregunta: ¿Cómo se evalúa 2 - 3 * 2 / 6 + 4 ?.....

- Podemos tener la expression: a = b = c, y se evalúa de derecha a izquierda a = (b = c).

  Y después de la operación, todas las variables tendrán el valor de c.

- Comentarios de bloque:

  /* hola como están

  espero que todos bien */

Más observaciones:

- Más operadores aritméticos:
  - x++, x--, ++x, --x : respectivamente suman o restan 1 a x
  - Pero utilizados en una expresión como: a = y*(x++), primero toma el valor inicial de x para evaluar a=y*x , y luego suma 1 a x
  - Operadores +=, -=, *=, /= y %=
    - Ejemplo: x *= y es equivalente a x = x* y
    - x += y * z es equivalente a x = x + (y * z), toma la expression completa del lado derecho.

Dado el programa:

```
#include <iostream>

int main() {
        int xx = 5, yy = 3, zz = 2;
        yy +=  xx++    +    zz;
        std::cout  << yy  << "  " <<  xx << '\n';
}
```

¿Qué imprime?:

10 6

¿Porqué cree que está errado el programa siguiente?:

```
#include <iostream>

int main() {
        int xx = 5, yy = 3, int zz = 2;
        yy +=  xx++    +    zz;
        std::cout  << yy  << "  " <<  xx << '\n';
}
```

Antes del último int debe ser ; y no ,

Hagamos juntos: ¿Qué imprime?:

```
int x1 = 2, y1, x2 = 2, y2, z;
y1 = ++x1; y2 = x2++; z = ++x1 + y2;
std::cout << x1 << " " << x2 << '\n';
std::cout << y1 << " " << y2 << '\n';
std::cout << x1 << " " << z << '\n';
```

```
int x1 = 2, y1, x2 = 2, y2, z;
y1 = ++x1; y2 = x2++; z = ++x1 + y2;
std::cout << x1 << " " << x2 << '\n';
std::cout << y1 << " " << y2 << '\n’;
std::cout << x1 << " " << z << '\n';
```

El tipo booleano:

```
int main() {
    // Declare some Boolean variables
    bool a = true, b = false;
    std::cout << "a = " << a << ", b = " << b << '\n';
    // Dar otro valor a a
    a = false;
    std::cout << "a = " << a << ", b = " << b << '\n';
    // Podemos asignar un entero cualquiera a un booleano y si es
    // distinto de cero lo convierte en 1 (true) , si no en false
    a = 1;       b = 0;
    std::cout << std::boolalpha; // instruir a la salida de imprimir true o false en lugar de 1 o 0
    std::cout << "a = " << a << ", b = " << b << '\n';
}
```

¿Qué imprime?:

a = 1, b = 0
a = 0, b = 0
a = true, b = false

Operadores relacionales para expresiones lógicas (booleanas):

Operator	Meaning
==	Equal to
<	Less than
>	Greater than
<=	Less than or equal to
>=	Greater than or equal to
!=	Not equal to

Table 5.1: C++ Relational operators

Expression	Value
`10 < 20`	always true
`10 >= 20`	always false
`x == 10`	true only if x has the value 10
`X != y`	true unless x and y have the same values

Table 5.2: Relational operator examples

La instrucción (o constructor) condicional:

```
if ( x == 0 )      // los paréntesis en la condición son obligatorios…..
        std::cout <<  "x es cero"  << "\n";
else
{
        std::cout <<  "x es " ;
        std::cout <<  x <<  " \n";
}
```

opcional

bloque de instrucciones entre llaves (instrucción compuesta)

Operadores y expresiones booleanas:

(x != 0) && ((z/x) > 1)

&& es el Y lógico. ¿Qué valor resulta de evaluar (2 == 3) && (2 != 3)? ….

|| es el O lógico ¿Qué valor resulta de evaluar (2 == 3) || (2 != 3)? ….

! es la negación de una expresión lógica: !(2 == 3) ¿qué valor de verdad tiene?

Cuidado con una expresión como la siguiente: 1 <= x <= 10

C++ la evaluaría (1 <= x) <= 10 si x es mayor o igual a 1, la expresión siempre sería verdad aunque x fuese igual a 20 porque (1 <= x) es 1 (true)

Deberíamos colocar (1 <= x) && (x <= 10)

Hagamos juntos :

Hacer el programa en C++ siguiente

Leer dos números por pantalla x e y

Si el valor absoluto de x es menor o igual al valor absoluto de y entonces imprimir:

“el valor absoluto de x es menor o igual al valor absoluto de y”

Si no imprimir:

“el valor absoluto de x es mayor que el valor absoluto de y”

Ojo: imprimir los valores de x e y dentro de la frase

```
#include <iostream>
int main(){
    double x,y;
    std::cout << "Coloque dos números: ";
    std::cin>> x >> y; std::cout <<std::endl<< x <<" "<< y <<std::endl;
    double z = ((x>=0)?x:-x),w =  ((y>=0)?y:-y);
    if (z<=w) std::cout << "valor absoluto de " << x << " menor o igual a "<<
           "valor absoluto de " << y;
    else std::cout << "valor absoluto de " << x << " mayor que "<<
    "valor absoluto de " <<y;
}
```

Tambien se puede hacer sin utilizar dos variables adicionales....

Precedencia de los operadores en C++ de la más alta a la más baja (se los dejo para que lo vean en su casa)

Arity	Operators	Associativity
unary	(post) ++, (post) --, `static_cast`	
unary	(pre) ++, (pre) --, !, +, -	
binary	*, /, %	left
binary	+, -	left
binary	<<, >>	left
binary	>, <, >=, <=	left
binary	==, !=	left
binary	&&	left
binary	||	left
binary	=, +=, -=, *=, /=, %=	right

Hagamos juntos un programa que lea un número entre 0 y 15 (que posee 4 dígitos en su representación binaria) y luego imprima una cadena de 4 caracteres (string) de “x” o “y” de longitud 4 dependiendo de si el dígito en su representación binaria es 1 ó 0 respectivamente.

Ejemplo: Si lee 7 su representación binaria es 0111 imprime yxxx ,
Si lee 1 su representación binaria es 0001 imprime yyyx

Si el numero es mayor o igual a 8 dividimos entre 8 e imprimimos x como primer dígito de izquierda a derecha
Si es menor que 8 se imprime y
Y asi sucesivamente, entre 4 luego entre 2

El programa debe validar primero que el número esté entre 0 y 15

Hagamos juntos un programa que lea un número entre 0 y 15 (que posee 4 dígitos en su representación binaria) y luego imprima una cadena de 4 caracteres (string) de “x” o “y” de longitud 4 dependiendo de si el dígito en su representación binaria es 1 ó 0 respectivamente.

```
#include <iostream>
 int main() {
      int valor;
      // Obtener numero
      std::cout << "Coloque un numero entre 0 y 15: ";
      std::cin >> valor;
      // el entero debe estar entre 0 y 15
      if ((0 <= valor) &&  (valor <= 15))  {
          if (valor >= 8)  {
               std::cout << 'x';
               valor %= 8;
          }
          else  std::cout << 'y';

          if (valor >= 4)  {
               std::cout << 'x';
               valor %= 4;
          }
          else     std::cout << 'y';
          if (valor >= 2)  {
               std::cout << 'x';
               valor %= 2;
          }
          else std::cout << 'y';
          if (valor == 1) std::cout << 'x';
          else std::cout << 'y';
          std::cout << '\n';
      }
}
```

Instrucciones iterativas ó bucles:

while (expresión booleana)

bloque de instrucciones

siguiente instrucción

Instrucciones iterativas ó bucles:

Ejemplo: Sumar los primeros 10 números naturales 1+2+3+… +10

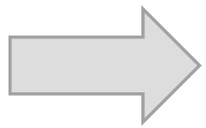
```
int i =1 , sum = 0, n=10;
while( i <= n){
    sum = sum + i;
    i = i + 1;
}
```

Instrucciones iterativas ó bucles:

```
do
   bloque de instrucciones (igual entre llaves)
while( expresión);
siguiente instrucción
```

¿Cómo expresar un while con un do?:

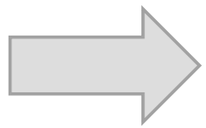
```
int i = 1, sum = 0, n=10;
while( i <= n)
    sum = sum + i++;
```



```
int i = 1, sum = 0, n=10;
 if ( i<=n)
       do
          sum = sum + i++;
      while (i<=n);
```

¿Cómo expresar do con un while?:

```
int i = 1, sum = 0, n=10;
 do
     sum = sum + i++;
while (i<=n);
```

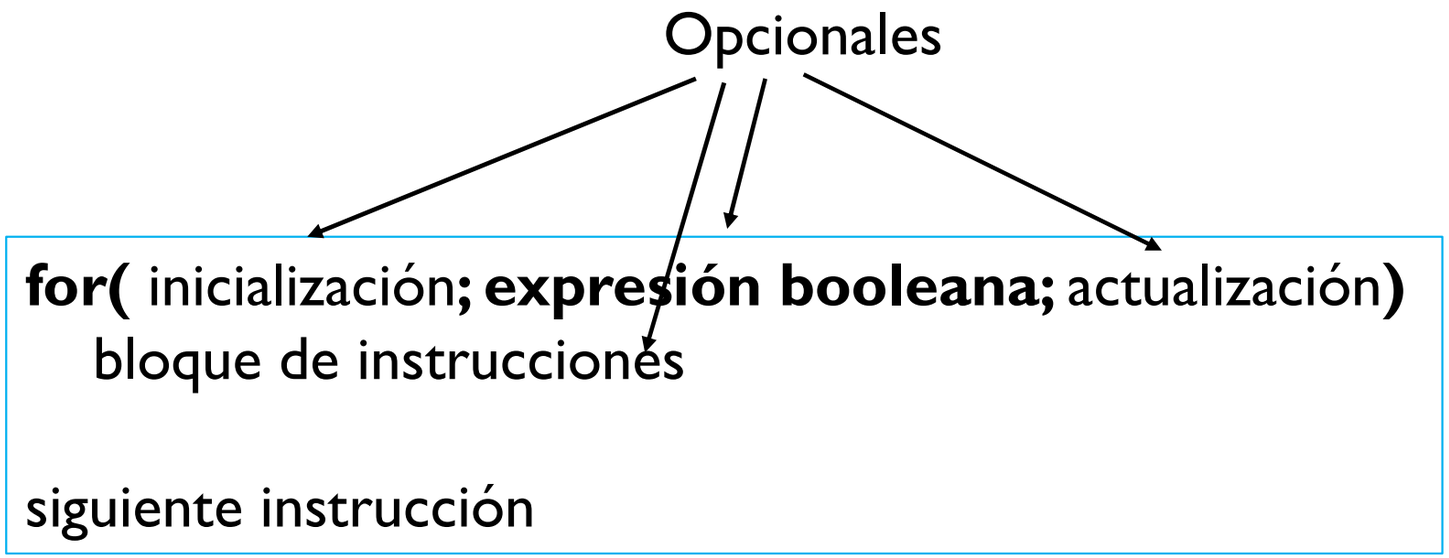


```
int i = 1, sum = 0, n=10;
bool a = true;
 while (a || ( i<=n) ){
     sum = sum + i++;
    a = false;
}
```

Mismo ejemplo anterior de transformar la representación binaria en x (1) e y(0) pero solicitando rectificar en caso de que el numero esté fuera del rango

```
#include <iostream>
 int main() {
    int valor;
    // Obtener numero
     while (true)  {
        std::cout << "Coloque un numero entre 0 y 15: ";
        std::cin >> valor;
        // Integer must be less than 1024
        if ((0 <= valor) &&  (valor < 16))  {
            if (valor >= 8)  {
                std::cout << 'x';
                valor %= 8;
            }
            else  std::cout << 'y';
                                                    if (valor >= 4)  {
                                                        std::cout << 'x';
                                                        valor %= 4;
                                                    }
                                                    else      std::cout << 'y';
                                                    if (valor >= 2)  {
                                                        std::cout << 'x';
                                                        valor %= 2;
                                                    }
                                                    else std::cout << 'y';
                                                    if (valor == 1) std::cout << 'x';
                                                    else std::cout << 'y';
                                                    std::cout << '\n';
                                                    break;
                                                }
                                                else std::cout << "Colocar num entre 0 y 15" << '\n';
                                            }
                                        }
```

La instrucción **for**:



Ejemplos:

```
// imprimir todos los pares (j,i) tales que j <= i <= 10

for( int i = 1; i <= 10; i++ )  // alcance de i solo
                                //  en el bloque del for
   for( int j = 1; j <= i; j++ )
      std::out <<  " ( " << j << " , " <<  i << " ) "
               << std::endl;
```

Otros ejemplos:

```
int i = 0;
for(  ; i <= 5;  )
    std::cout << i++ << '\n';
```

```
int i=0,sum=0;
for(  ; i <= n;    i++)
    sum += n ;
```

Discutamos juntos como haríamos un programa en C++ que lea un número entero no negativo N y luego imprima todos los números primos menores o iguales a N . (Recuerde que un número primo es un entero mayor que 1 divisible solo por el mismo y por 1)

Para probar que un número M es primo basta probar que no es divisible por los números enteros que van desde 2 hasta la raíz cuadrada de M. ¿Por qué hasta raíz?

Respuesta: Si M admite un factor (no es primo), es decir M = AxB y A es <= B entonces A es <= R (la raíz de M)
Porque si A fuese mayor que R, como B es >= A entonces AxB sería mayor que RxR que es igual a M, una contradicción porque AxB es igual a M

Por lo tanto basta probar que M no tiene factores entre 2 y R

Podemos usar sqrt(x) para determinar la raíz cuadrada. Está en la biblioteca <cmath>

Ejemplo: Imprimir los números primos hasta un entero leído por pantalla

```
#include <iostream>
#include <cmath>

int main() {
    int N;
    double raiz_de_valor;
    std::cout << "Imprimir primos hasta que valor? ";
    std::cin >> N;
    for (int valor = 2; valor <= N; valor++) {
        // se comprueba si valor es primo
        bool es_primo = true;  /* Inicialmente suponemos
            que es primo para entrar al for e ir verificando */
        raiz_de_valor = sqrt(valor);
        for (int factor_de_prueba = 2;
              es_primo && (factor_de_prueba <= raiz_de_valor);
              factor_de_prueba++)
            es_primo = (valor % factor_de_prueba != 0);
        if (es_primo)
            std::cout << valor <<  " ";   // Imprime un primo
    }
    std::cout << '\n';  // Ir a la siguiente línea
}
```

Vemos que podíamos utilizar una **función** de la biblioteca "**cmath**" de C++ para determinar la raíz cuadrada de un número: sqrt ( ).

¿Cómo funciona una llamada a una función?

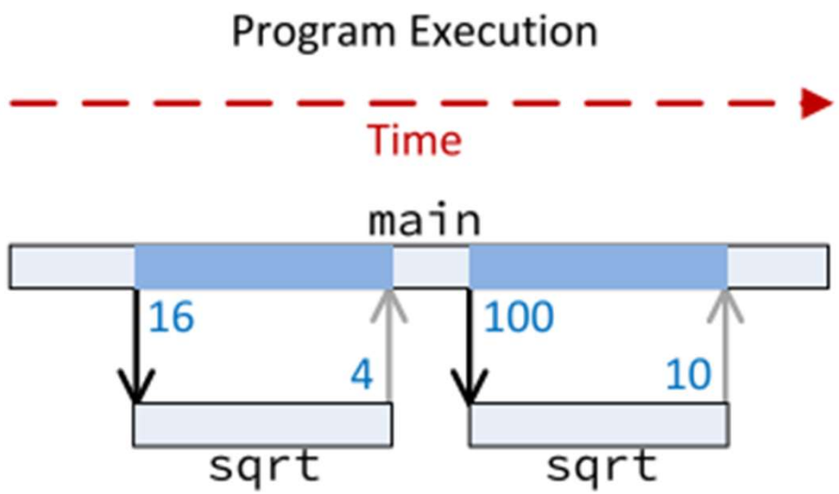
```
int main() {
    double value;

    //  Assign variable
    value = 16;

    //  Compute s square root
    double root = sqrt(value);

    //  Compute another
    root = sqrt(100);
}
```

parámetro



Desde el punto de vista del usuario, al utilizar una función el usuario necesita conocer:

- Su nombre: **sqrt**
- Los tipos de los parámetros de la función: **double**
- El tipo del resultado de la función: **double**

Esta información constituye el prototipo de la función:

double sqrt (double)

La función **sqrt( )** viene en tres formas:

- double sqrt(double)
- float sqrt(float)
- long double sqrt(long double)

El nombre de la función es la misma pero cambia el tipo de los parámetros. Son tres versiones distintas de la misma función pero para parámetros de diferente tipo.

Decimos que la función **sqrt( )** está "sobrecargada" (overloaded). Luego veremos este concepto en más detalle

Hay funciones que no necesitan parámetros o que no devuelven ningún valor de retorno:

void exit (int): detiene la ejecución del programa y devuelve un número al sistema operativo para saber si el programa terminó normalmente o no. La palabra reservada "void" indica que la función no devuelve ningún valor.

La función:
int rand( ): genera un número aleatorio entre 0 y RAND_MAX (una constante)

Las dos están en la biblioteca <cstdlib>

Programa que utiliza las funciones **clock( ) y sqrt( )** de la biblioteca estándar de C++ para determinar el tiempo que tarda un programa en calcular los primeros 1000 números primos:

```
#include <iostream>
#include <ctime>
#include <cmath>

/* Imprimir los primeros 1000 números primos y al final
   imprimir el tiempo que tardó el programa */

int main() {
    clock_t tiempo_com = clock(), // tiempo de comienzo
            tiempo_fin;  // tiempo de finalizacion

    for (int valor = 2; valor <= 1000; valor++) {
        // ver si "valor" es primo
        bool es_primo = true;  // Suponemos que es primo
        // probar todos los factores de 2 a sqrt(valor)
        for (int factor_prueba = 2;
                es_primo && factor_prueba <= sqrt(valor);
                factor_prueba++)
            es_primo = (valor % factor_prueba != 0);
        if (es_primo)
            std::cout << valor <<  " ";   // Imprimir primo
    }
    std::cout << '\n';  // ir a la linea siguiente
    tiempo_fin = clock();
    // Imprimir tiempo transcurrido
    std::cout << "Tiempo transcurrido en segundos: "
        << static_cast<double>(tiempo_fin -
            tiempo_com)/CLOCKS_PER_SEC
        << " seg." << '\n';
}
```

Podemos escribir nuestras propias funciones en C++. Escribamos una función que calcula la raíz cuadrada por un método conocido como el método de Newton

```
#include <cmath>
 // Calcula la raiz cuadrada de un numero x
 double raiz_cuadrada(double x) {
    double diff;
    // Raiz provisional
    double root = 1.0;
    do { // Iterar hasta conseguir una
         // aproximacion de 4 decimales
        root = (root + x/root) / 2.0;
        // Determinar la aproximacion
        diff = root * root - x;
    } while (diff > 0.0001 || diff < -0.0001);
    return root;
 }
```

```
int main() {
    // Comparar nuestra funcion con la
    // de la biblioteca estandar para los
    // reales entre 1 y 10, con paso de 0.5
    for (double d = 1.0; d <= 10.0;
            d += 0.5)
       std::cout << std::setw(7)
                 << raiz_cuadrada(d)
                 << " : " << sqrt(d) << '\n';
}
```

diff: variable local a la función

Valor absoluto de diff mayor que 0.0001)

Parámetros formales (uno en este caso)

Parámetros reales (uno en este caso)

Los parámetros reales de una llamada a función puede ser cualquier expresión cuyo tipo sea el mismo del parámetro formal correspondiente:

Ejemplos:

- sqrt(2.6 + 4.5)
- sqrt (max (x, y)) : suponemos que **max** es una función que devuelve el máximo entre dos números. Las variables x e y deben ser números
- sqrt (sqrt (16))

- En el caso de la función **sqrt( )** el parámetro se pasa **POR VALOR**, es decir que el **valor** del parámetro real es COPIADO al parámetro formal. Si **sqrt( )** modificara su parámetro formal en su cuerpo, NO se modifica el valor del parámetro real

```
void xx (int x){
    x++;          //x es local a xx ( )
    std::cout << x << std::endl;
}
 int main() {
    int x = 2;  // x es local a main( )
    xx(x);
    std::cout << x << std::endl;
}
```

¿Qué imprimiría?:
3
2

Es importante documentar los programas y en especial las funciones (**Esto será exigido y calificado en los proyectos**).

Al comienzo de la definición de una función se debe colocar un comentario que describe la naturaleza de la función:

- Describir qué hace la función, el propósito de la función.
- Describir cada parámetro formal.
- Describir que devuelve la función.

Ejemplo de documentación de una función:

```
/*
 * distancia(x1, y1, x2, y2)
 * Descripción: Calcula la distancia entre dos puntos del plano cartesiano
 * x1 es la coordenada x del primer punto
 * y1 es la coordenada y del primer punto
 * x2 es la coordenada x del segundo punto
 * y2 es la coordenada y del segundo punto
 * Devuelve la distancia entre (x1,y1) y (x2,y2)
 */

double distancia(double x1, double y1, double x2, double y2) { ... }
```

Ejercicios:

En el primer módulo, en Módulos de Módulo 7 podrán encontrar una lista detalladas de los ejercicios por temas.

Hay muchos ejercicios en los libros de Vicente Benjumea, de Luis Joanes y de Halterman, los cuales están los PDF en el aula

PREGUNTAS???