Laboratorio de Programación de Algoritmos y Estructura de Datos I

Algoritmos y Estructuras de Datos I

Departamento de Computación, FCEyN, Universidad de Buenos Aires.

Bienvenida!!!

- Equipo docente
  - Brian Curcio (JTP)
  - Pablo Negri (JTP)
  - Gustavo Landfried (Ay1)
  - Gonzalo Fernández (Ay2)
  - Eric Brandwein (Ay2)
- Aula o Laboratorio
  - Laboratorios 01 y 06 (18-03)
  - Laboratorios 01 y 04 (25-03)
  - Laboratorios 01 y Turing (en adelante)
  - Aula: 2
- Vía de comunicación con la cátedra
  - Campus Virtual Exactas (https://campus.exactas.uba.ar/)

Objetivos del Laboratorio

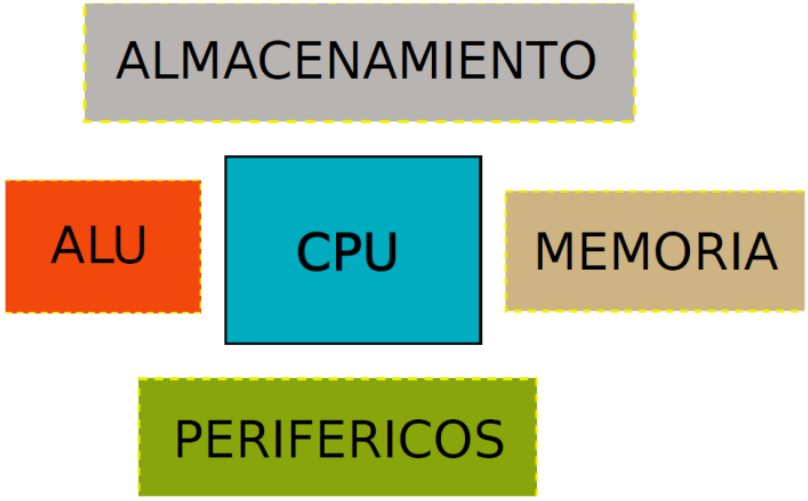
- Dar los primeros pasos en programación imperativa (C++)
- Familiarizarse con una IDE (CLION)
- Programar algoritmos de complejidad creciente
- Hacer Debugging de sus programas (CLION)
- Generar Test de Funcionalidad de sus programas (GTEST)

Régimen de aprobación

- Parciales
  - Dos parciales (requerimiento para rendir: LU con Prácticos de Algebra I firmados)
  - Dos recuperatorios (al final de la cursada)
- Trabajos prácticos Obligatorios
  - Dos entregas TPE+TPI
  - Dos recuperatorios.
  - Grupos de DOS alumnos.
  - Mini-TPE
- Examen final o un coloquio (en caso de tener dado el final de Álgebra I al finalizar la cursada)

Comencemos por el inicio...

De que elementos está compuesta una computadora?



- Hardware
- Software
  - Sistema Operativo
  - Programas de Usuario

Lenguajes de Programación

Se dividen en tres tipos:

1. Lenguajes de máquina
2. Lenguajes ensambladores
3. Lenguajes de alto nivel

Lenguajes de Programación

Se dividen en tres tipos:

1. **Lenguajes de máquina**
2. Lenguajes ensambladores
3. Lenguajes de alto nivel

Son cadenas de números. No requieren traducción para la máquina.

+1300042774
+1400245324
+1200463450

Lenguajes de Programación

Se dividen en tres tipos:

1. Lenguajes de máquina
2. **Lenguajes ensambladores**
3. Lenguajes de alto nivel

Operaciones elementales. Los *ensambladores* traducían estas instrucciones en código de máquina en tiempo real.

```
load sueldoBase
add sueldoExtra
store sueldoBruto
```

Lenguajes de Programación

Se dividen en tres tipos:

1. Lenguajes de máquina
2. Lenguajes ensambladores
3. **Lenguajes de alto nivel**

Instrucciones de más alta complejidad. Precisan un traductor o *compilador* para su conversión en código de máquina.

```
sueldoBruto = sueldoBase
+ sueldoExtra
```

Fases de ejecución de un programa

1. EDITOR: El programador crea el programa y lo guarda en el disco (extensión .cpp).
2. PREPROCESADOR: Se pre-procesa el código verificando si existen errores.
3. COMPILADOR: Se produce la traducción al código **objeto** y se almacena en el disco (extensión .obj).
4. ENLAZADOR: Se relacionan con el código objeto las bibliotecas necesarias para el funcionamiento del programa y se crea el archivo ejecutable (extensión .exe).
5. CARGADOR: Se "coloca" el programa en la memoria.
6. EJECUCIÓN: La CPU ejecuta secuencialmente las instrucciones del programa.

Paradigmas de lenguajes de programación

- **Paradigma:** Definición del modo en el que se especifica el cómputo (que luego es implementado a través de programas).
  1. Representa una "toma de posición" ante la pregunta: ¿cómo se le dice a la computadora lo que tiene que hacer?.
  2. Todo lenguaje de programación pertenece a un paradigma.

- Estado del arte:
  1. Paradigma de programación imperativa: C, Basic, Ada, Clu
  2. Paradigma de programación en objetos: Smalltalk
  3. Paradigma de programación orientada a objetos: C++, C#, Java
  4. Paradigma de programación funcional: LISP, F#, Haskell
  5. Paradigma de programación en lógica: Prolog

Un poco de historia: El Lenguaje C++

- El lenguaje C fue creado por Dennis Ritchie entre 1969 y 1973 en Bell Labs, para una reimplementación de Unix.
- El lenguaje C++: fue creado por Bjarne Stroustrup en 1983.
- Etimología: C $\rightarrow$ new C $\rightarrow$ C with Classes $\rightarrow$ C++.
- Lo usaremos como lenguaje imperativo aunque también soporta parte del paradigma de objetos (+Algo2).
- Hay varios standards (C++98, C++11, C++14). Usaremos **C++11**.

Programación imperativa

- **Entidad fundamental:** variables, que corresponden a posiciones de memoria (RAM) y cambian explícitamente de valor a lo largo de la ejecución de un programa.
- **Operación fundamental:** asignación, para cambiar el valor de una variable.

  1. Una variable no cambia a menos que se cambie explícitamente su valor, a través de una asignación.
  2. Las asignaciones son la única forma de cambiar el valor de una variable.
  3. En los lenguajes tipados (*typed*), las variables tienen un tipo de datos y almacenan valores del conjunto base de su tipo.

Programación Imperativa: Variables

- Para almacenar valores utilizamos variables, que se declaran con un tipo de datos asociado:

```
1  #include <iostream>
2
3  int main() {
4      int a = 11;
5      std::cout << a;
6      return 0;
7  }
```

- A partir de la línea 5, la variable **a** contiene el entero 11.
- En el siguiente comando, se accede a esta variable y se imprime por consola su valor.

Tipos de datos de C++

- Un **tipo de datos** es ...
  1. ... un conjunto de valores (llamado el *conjunto base* del tipo),
  2. ... junto con una serie de operaciones para trabajar con los elementos de ese conjunto.
- En C++ tenemos tipos de datos que implementan (en algunos casos parcialmente) cada uno de los tipos de datos del lenguaje de especificación:
  - El tipo `int` para números enteros ($\mathbb{Z}$)
  - El tipo `float` para números reales ($\mathbb{R}$)
  - El tipo `bool` para valores booleanos (Bool)
  - El tipo `char` para caracteres (Char)
- Atención: Ni `int` ni `float` contienen todos los valores de $\mathbb{Z}$ y $\mathbb{R}$, pero a los fines de AED1, podemos asumir que $\mathbb{Z} =$ `int` y $\mathbb{R} =$ `float`.

Operadores aritméticos

- Asociados a los tipos de variables, se definen los siguientes operadores aritméticos:
  1. + y -: suma y resta.
  2. *, /: multiplicación y división.
  3. **%**: módulo, devolviendo el resto de la división entre dos números.

Concordancia de tipos

- En C/C++ es obligatorio asignar a cada variable una expresión que coincida con su tipo, o que el compilador sepa cómo convertir en el tipo de la variable.
- Se dice que C++ es un lenguaje débilmente tipado.

```
1  int main() {
2    int a = "Hey, hey!"; // No! La expresion asignada no es un int
3    ...
4  }
```

Declaración y asignación de variables

- TODAS LAS VARIABLES se deben declarar antes de su uso.
  1. **Declaración:** Especificación de la existencia de la variable, con su tipo de datos.
  2. **Asignación:** Asociación de un valor a la variable, que no cambia a menos que sea explícitamente modificado por otra asignación.
  3. **Inicialización:** La primera asignación a una variable. Entre la declaración y la inicialización tiene "basura".

```
1  int main() {
2    int a; // Declaracion, aqui a no tiene valor util
3    a = 5; // Inicializacion
4    a = a+2; // Asignacion de un nuevo valor
5    ...
6  }
```

- Una variable puede ser inicializada al declararla: `int a = 5` es válido.

Expresiones en C++

- El elemento del lado derecho de una asignación es una expresión.
- Esta expresión también puede incluir llamadas a funciones:
-

```
1  #include <iostream>
2  #include <cmath> // incluye seno, coseno, tangente, etc.
3
4  int main() {
5     float x = 2 + 5;
6     float y = sin(x) + cos(x);
7
8     std::cout << y;
9     return 0;
10 }
```

Operadores de Comparación e Igualdad

- Existen operadores de comparación e igualdad que devuelven un resultado booleano:
  1. == y !=: igualdad y desigualdad.
  2. $>$, $<$, $\geq$, $\leq$: mayor, menor, mayor e igual, menor e igual.

- Las comparaciones pueden realizarse sobre constantes numéricas o sobre variables.

```
1    (5 == 5); // devuelve true
2    (6 <= 2); // devuelve false
3    (b == d); // depende de las varialbes b y d
4    (c > a);
5  }
```

Operadores Lógicos

- Los operadores lógicas en C++ son:
  1. `&&`: AND lógico.
  2. `||`: OR lógico.
  3. `!` : NOT lógico.

- Veamos un ejemplo utilizando variables y operaciones lógicas.

Ejemplo de Operaciones Lógicas

- Crear un nuevo proyecto `File --> New Project`.
- Llamarlo, por ejemplo LogicOp.
- Reemplazar el main.cpp con el siguiente código:
-

```
1  #include <iostream>
2
3  int main() {
4
5      bool a = false;
6      bool b = true;
7      bool c;
8
9      c = a && b;
10
11     std::cout << "Valor c: " << c << std::endl;
12     return 0;
13 }
```

Ejemplo de Operaciones Lógicas

- Al ejecutarlo, el panel de salida muestra:

```
variable c: 0

Process finished with exit code 0
```

- El valor de la variable c es 0!!!
- Y el valor booleano????
- Podemos imprimir un mensaje más adecuado?.

Ejemplo de Operaciones Lógicas

- Introducimos el Operador Condicional Ternario
- `expresion ? resultado1 : resultado2` (les hace recordar de la teoria?)
- Reemplazar el código
-

```
1 std::cout << "Valor c: " << (c ? "true" : "false") << std::endl;
```

- Tambien puede realizar otro tipo de operaciones interesantes
-

```
1 std::cout << "El maximo es: " << (a > b ? a : b) << std::endl;
```

Operadores Lógicos

- Los operadores && y || utilizan lógica de cortocircuito: No se evalúa la segunda expresión si no es necesario.
- En otras palabras && implementa el $\wedge_L$ y || implementa el $\vee_L$
-

```
1 // inversoMayor entre n y m
2 bool c = (n != 0 && 1/n > m);
3 std::cout << "El inverso de n " << (c ? "" : " no ");
4 std::cout << " es mayor que m" << std::endl;
```

- Si $n = 0$, entonces el primer término es falso, pero el segundo está indefinido! En C/C++, esta expresión evalúa directamente a falso.
- Solamente se evalúa $1/n > m$ si $n \neq 0$.

Entrada y Salida desde Consola

- En los sistemas UNIX la consola (teclado y monitor) permitía interactuar con una aplicación
- `cout`: console out. Imprime por pantalla un dato (usamos $<<$)
- `cin`: console in: Lee un dato del teclado (usamos $>>$ y hay que apretar ENTER)

Demo #2: Entrada y Salida desde Consola

¿Qué pasa al ejecutar este programa?

```
1  #include <iostream>
2
3  int main() {
4      cout << "Ingrese un valor entero" << endl;
5      int valor = 0;
6      std::cin >> valor;
7      std::cout << "El valor ingresado fue " << valor << std::endl;
8      return 0;
9  }
```

Demo #3: Expresiones indefinidas en C++

¿Qué pasa al ejecutar este programa?

```
1  #include <iostream>
2
3  int main() {
4    std::cout << "Ingrese un valor por teclado " << std::endl;
5    int x;
6    std::cin >> x;
7    int r = 1 /x;
8    std::cout << "Su inverso multiplicativo es " << r << std::endl;
9    return 0;
10 }
```

- ¿Qué valor retorna si ingresamos 10?
- ¿Qué valor retorna si ingresamos 0?

Expresiones indefinidas en C++

- Lamentablemente, C++ no define qué ocurre cuando evaluamos una operación indefinida
- Algunas cosas que pueden pasar
  - Termina la ejecución con un `exit code` distinto de 0
  - Continua la ejecución con un valor cualquiera (El HORROR, El HORROR, El HORROR)

Secuencialidad de la ejecución

- Los lenguajes imperativos son secuenciales: los comandos se ejecutan en orden, de arriba hacia abajo.

```
1  #include <iostream>
2  using namespace std;
3
4  int main() {
5     int a = 0;
6     int b = 0;
7     cout << "Ingrese un valor de a " << endl;
8     cin >> a;
9     cout << "Ingrese un valor de b " << endl;
10    cin >> b;
11    int d = 2*a + b;
12    int r = 3*b - (d*a);
13    cout << "El valor de r es " << r << endl;
14    return 0;
15 }
```

- La ejecución de un programa imperativo tiene temporalidad.

Recap: C++

Hasta ahora hemos visto:

- función `main` (punto de entrada)
- Librerías (Bibliotecas): `#include <...>`
- `cout`: salida por pantalla
- Tipos de datos: int, bool, char, float
- Declaración, inicialización y asignación de variables
- Ejecución secuencial de sentencias (ordenes/instrucciones)
- IDE: CLion

Estructuras de control

- La asignación es el único comando disponible para modificar el valor de una variable.
- El resto de las construcciones del lenguaje permite estructurar el programa para combinar asignaciones en función del resultado esperado. Se llaman estructuras de control:
  1. Llamados a Funciones
  2. Alternativas
  3. Ciclos

Declarando funciones en C++

- Podemos declarar nuestras propias funciones. Para eso debemos especificar:
  1. Tipo de retorno (puede retornar vacío, `void`)
  2. Nombre (obligatorio)
  3. Argumentos (o parámetros) (puede ser vacía)
- Los argumentos se especifican separados por comas, y cada uno debe tener un tipo de datos asociado.
- Cuando se llama a la función, el código "llamador" debe respetar el orden y tipo de los argumentos.

Funciones con valores de retorno

- Una función retorna un valor mediante la sentencia **return**.
- Por ejemplo, la siguiente función toma un parámetro entero y devuelve el *siguiente* valor:
-

```
1  int siguiente(int a) {
2    return a+1;
3  }
```

- Otra versión (quizás menos intuitiva):
-

```
1  int siguiente(int a) {
2    int b =0;
3    b = a+1;
4    return b;
5  }
```

Llamados a Funciones

- Volviendo al ejemplo anterior, podemos hacer un llamado (invocación) a la función **siguiente()** dentro de nuestro programa:

```
1  #include <iostream>
2  using namespace std;
3
4  int siguiente(int a) {
5    return a+1;
6  }
7
8  int main() {
9    int a = 5;
10   int b = 2 * siguiente(a); // llamado a siguiente
11
12   cout << b;
13   return 0;
14 }
```

Funciones con n-argumentos

- En caso de que haya más de un parámetro, se separan por comas:

```
1  #include <iostream>
2  using namespace std;
3
4  int suma(int a, int b) {
5     return a+b;
6  }
7
8  int main() {
9     int a = suma(2,3); // llamado con 2 argumentos
10    cout << a;
11    return 0;
12 }
```

Instrucción alternativa

- Tiene la siguiente forma, donde B es una expresión lógica (que evalúa a **boolean**) y S_1 y S_2 son bloques de instrucciones:
-

```
1  if (B) {
2    S1
3  } else {
4    S2
5  }
```

- Se evalúa la guarda B. Si la evaluación da **true**, se ejecuta S_1. Si no, se ejecuta S_2.
- La rama positiva S_1 es obligatoria. La rama negativa S_2 es optativa.
- Si S_1 o S_2 constan de más de una instrucción, es obligatorio que estén rodeados por llaves.

Instrucción alternativa

- ¿Cómo podemos escribir un programa que calcule el valor absoluto de x?
-

```
1  int valorAbsoluto(int n) {
2      int res = 0;
3      if( n > 0 ) {
4          res = n;
5      } else {
6          res = -n;
7      }
8      return res;
9  }
```

Demo #6: Instrucción alternativa

```
1  #include <iostream>
2  using namespace std;
3
4  int valorAbsoluto(int n) {
5     int res = 0;
6     if( n > 0 ) {
7        res = n;
8     } else {
9        res = -n;
10    }
11    return res;
12 }
13
14 int main() {
15    int a = 0;
16    cout << "Ingrese valor: " << endl;
17    cin >> a;
18    int abs_a = valorAbsoluto(a);
19    cout << "Valor absoluto: " << abs_a << endl;
20    return 0;
21 }
```

Instrucción alternativa

- Podemos también hacer directamente "**return** res" dentro de las ramas de la alternativa.

```
1  int valorAbsoluto(int n) {
2    if( n > 0 ) {
3      return n;
4    } else {
5      return -n;
6    }
7  }
```

- Cuidado: **return** termina inmediatamente la ejecución de la función.

  $\Rightarrow$ Puede dejar las variables en un estado inconsistente!

Instrucción alternativa

- Los operadores && y || utilizan lógica de cortocircuito: No se evalúa la segunda expresión si no es necesario.
- En otras palabras && implementa el $\wedge_L$ y || implementa el $\vee_L$
-

```
1  boolean inversoMayor(int n, int m) {
2
3    if( n != 0 && 1/n > m ) {
4      return true;
5    } else {
6      return false;
7    }
8  }
```

- Si $n = 0$, entonces el primer término es falso, pero el segundo está indefinido! En C/C++, esta expresión evalúa directamente a falso.
- Solamente se evalúa $1/n > m$ si $n \neq 0$.

Instrucción alternativa

- La Instrucción alternativa tambien permite obviar el `else` si no se desea ejecutar ninguna instrucción:
- Por ejemplo:

```
1   ...
2   if( x>0 ) {
3       result = true;
4   } else {
5       // no hacer nada
6   }
7   ...
```

- Es equivalente a:

```
1   ...
2   if( x>0 ) {
3       result = true;
4   }
5   ...
```

Ciclos "while"

- Sintaxis:

```
while (B) {
      cuerpo del ciclo
}
```

- Se repite el cuerpo del ciclo mientras la guarda `B` se cumpla, cero o más veces. Cada repetición se llama una iteración.
- La ejecución del ciclo termina si no se cumple la guarda al comienzo de su ejecución o bien luego de ejecutar una iteración.
- Si/cuando el ciclo termina, el estado resultante es el estado posterior a la última instrucción del cuerpo del ciclo.
- Si el ciclo no termina, la ejecución nunca termina (se cuelga).

Ejemplo

```
1  int suma(int n) {
2     int i = 1;
3     int sum = 0;
4     while( i <= n ) {
5        sum = sum + i;
6        i = i + 1;
7     }
8     return sum;
9  }
```

Demo #7: instrucción while

```
1  #include <iostream>
2  using namespace std;
3
4  int suma(int n) {
5      int i = 1;
6      int sum = 0;
7      while( i <= n ) {
8          sum = sum + i;
9          i = i + 1;
10     }
11     return sum;
12 }
13
14 int main() {
15     int valorParaSumar = 0;
16     cout << "Ingrese valor para sumar: ";
17     cin >> valorParaSumar;
18     int resultadoDeSuma = suma(valorParaSumar);
19     cout << resultadoDeSuma;
20     return 0;
21 }
```

Ejemplo

- Estados al finalizar cada iteración del ciclo, para $n = 6$:

Iteración	i	suma
0	1	0
1	2	1
2	3	3
3	4	6
4	5	10
5	6	15

- Al final de las iteraciones (cuando se sale del ciclo porque no se cumple la guarda), la variable **sum** contiene el valor buscado.

Ejemplo

- La variable **i** se denomina la variable de control del ciclo.
  1. Cuenta cuántas iteraciones se han realizado (en general, una variable de control marca el avance del ciclo).
  2. En función de esta variable se determina si el ciclo debe detenerse (en la guarda).
  3. Todo ciclo tiene una o más variables de control, que se deben modificar a lo largo de las iteraciones.
- La variable **sum** se denomina el acumulador (o variable de acumulación) del ciclo.
  1. En esta variable se va calculando el resultado del ciclo. A lo largo de las iteraciones, se tienen resultados parciales en esta variable.
  2. No todo ciclo tiene un acumulador. En algunos casos, se puede obtener el resultado del ciclo a partir de la variable de control.

Ciclos "for"

- La siguiente estructura es habitual en los ciclos:
  1. Inicializar la variable de control.
  2. Chequear en la guarda una condición sencilla sobre las variables del ciclo.
  3. Ejecutar alguna acción (cuerpo del ciclo).
  4. Modificar en forma sencilla la variable de control.

- Para estos casos, tenemos la siguiente versión compacta de los ciclos, llamados ciclos "for".

-
```
1  int sum = 0;
2  for(int i=1; i<=n; i=i+1) {
3    sum = sum + i;
4  }
```

Otro ejemplo usando “for”

Podemos escribir un programa que indique si un numero $n \geq 2$ es primo usando ciclos y la instrucción `for`:

```
1  bool esPrimo(int n) {
2      int divisores = 0;
3      for(int i=2; i<n; i=i+1) {
4          if( n % i == 0 ) {
5              divisores = divisores + 1;
6          } else {
7              // no hacer nada
8          }
9      }
10     if (divisores == 0) {
11         return true;
12     } else {
13         return false;
14     }
15 }
```

Demo #8: Ejemplo de uso de for

```
1  #include <iostream>
2  using namespace std;
3
4  bool esPrimo(int n) {
5      int divisores = 0;
6      for(int i=2; i<n; i=i+1) {
7          if( n % i == 0 ) {
8              divisores = divisores + 1;
9          } else {
10             // do nothing
11         }
12     }
13     if (divisores == 0) {
14         return true;
15     } else {
16         return false;
17     }
18 }
```

```
1  int main() {
2      cout << "Ingrese numero:" << endl;
3      int a = 0;
4      cin >> a;
5      bool soyPrimo = esPrimo(a);
6      cout << "Primo: ";
7      if (soyPrimo) {
8          cout << "SI" << endl;
9      } else {
10         cout << "NO" << endl;
11     }
12     return 0;
13 }
```

Recursión en C/C++

- Podemos hacer llamados recursivos en C/C++!
- Sin embargo, el modelo de cómputo es imperativo, y entonces la ejecución es distinta en este contexto.
-

```
1  int suma(int n) {
2    if( n == 0 ) {
3      return 0;
4    } else {
5      return n + suma(n−1);
6    }
7  }
```

- Se calcula primero `suma(n-1)`, y hasta que no se tiene ese valor no se puede continuar la ejecución (orden aplicativo).
- No existe en los lenguajes imperativos el orden normal de los lenguajes funcionales!

¿Por qué un nuevo paradigma?

- Si podemos hacer recursión pero no tenemos orden normal, ¿por qué existen los lenguajes imperativos?
  1. La performance de los programas implementados en lenguajes imperativos suele ser muy superior a la de los programas implementados en lenguajes funcionales (la traducción al hardware es más directa).
  2. En muchos casos, el paradigma imperativo permite expresar algoritmos de manera más natural.
- Aunque los lenguajes imperativos permiten implementar funciones recursivas, el mecanismo fundamental de cómputo no es la recursión.

CLion

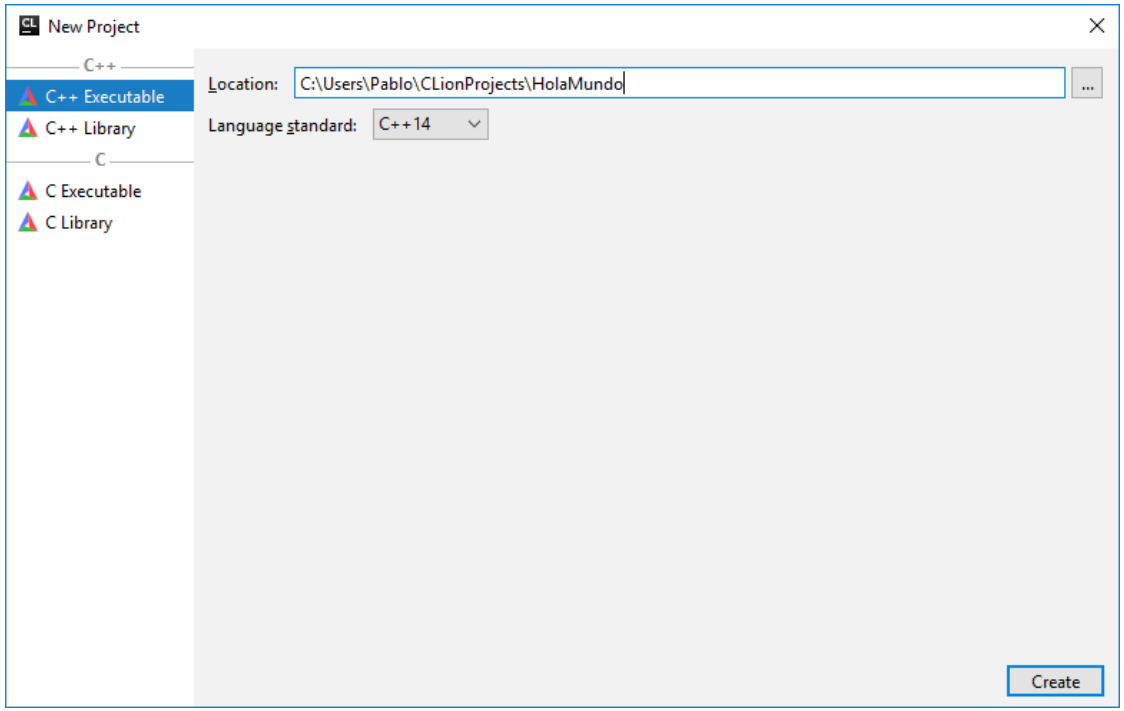
- En el Taller de Algo-1 vamos a utilizar un entorno de desarrollo de C++ (IDE) denominado CLion.
- Esta IDE permite crear proyectos, editar archivos, compilar, *debuggear*, entre otras cosas.
- Es una herramienta que se puede descargar bajo licencia estudiantil, y está instalada en las PCs del laboratorio.

Pantalla de bienvenida del CLion

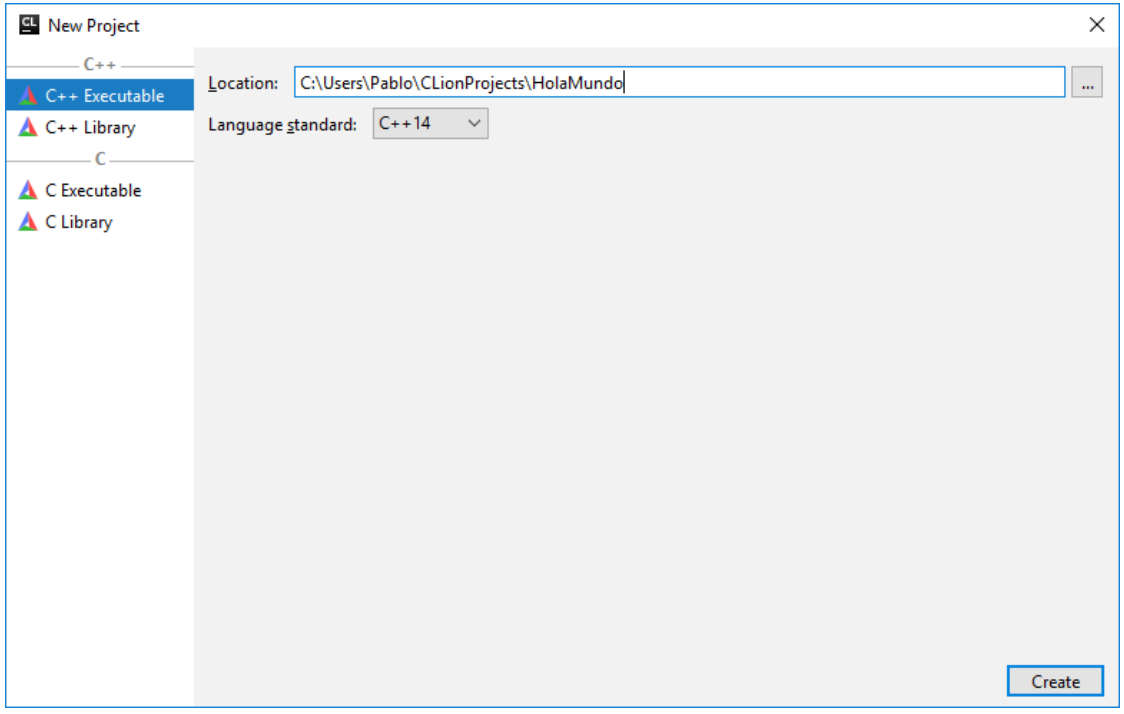
- Para lanzar el CLion en las PCs del laboratorio, abrir una consola (boton derecho del mouse sobre el escritorio), y escribir:

  `clion.sh`

- La ventana de inicio puede mostrar:



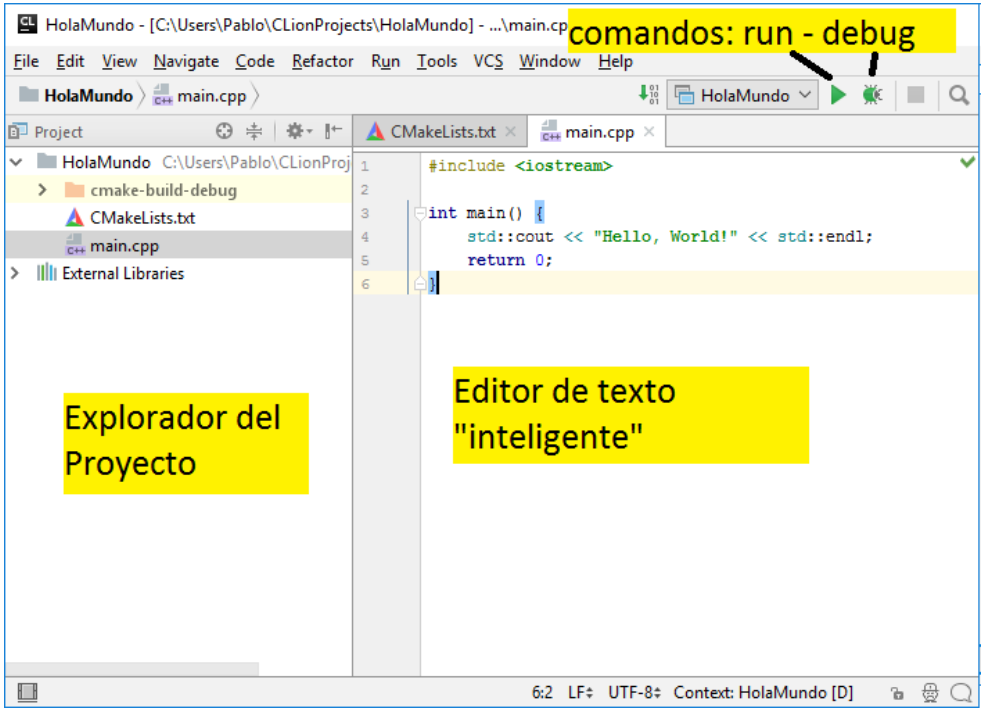
Pantalla de bienvenida del CLion



- La ventana de nuevo proyecto permite seleccionar el directorio de destino y el tipo de aplicación.
- En el ejemplo, el directorio destino es "HolaMundo", y el tipo de aplicación "C++ Executable".

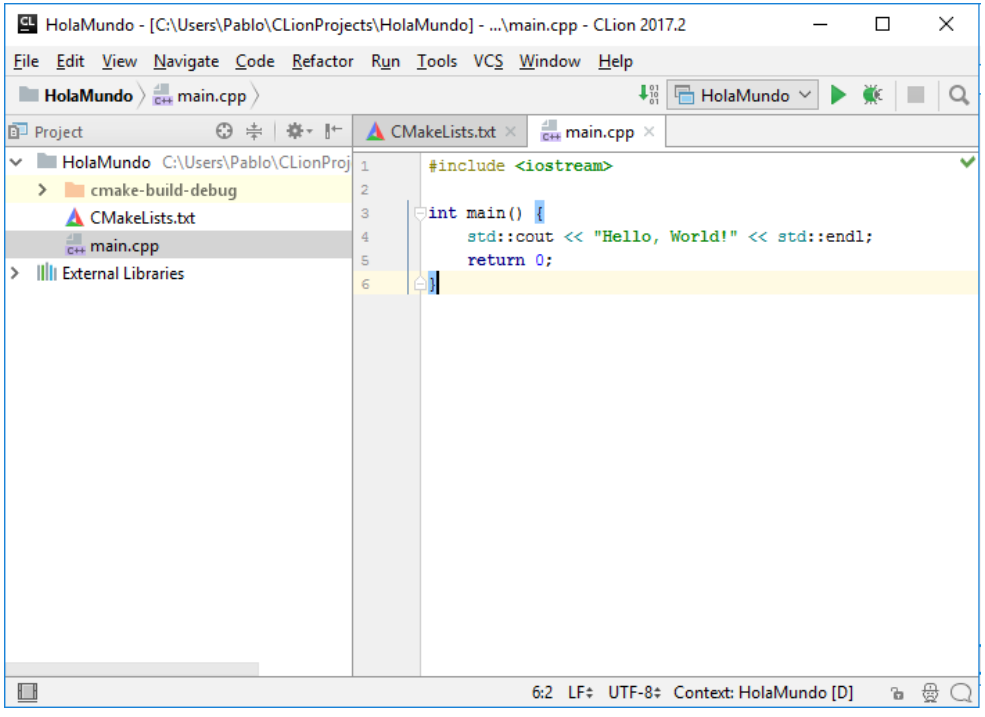
Proyecto Hola Mundo

- La IDE contiene dos paneles principales:
  1. Explorador de los archivos del proyecto
  2. Editor de texto

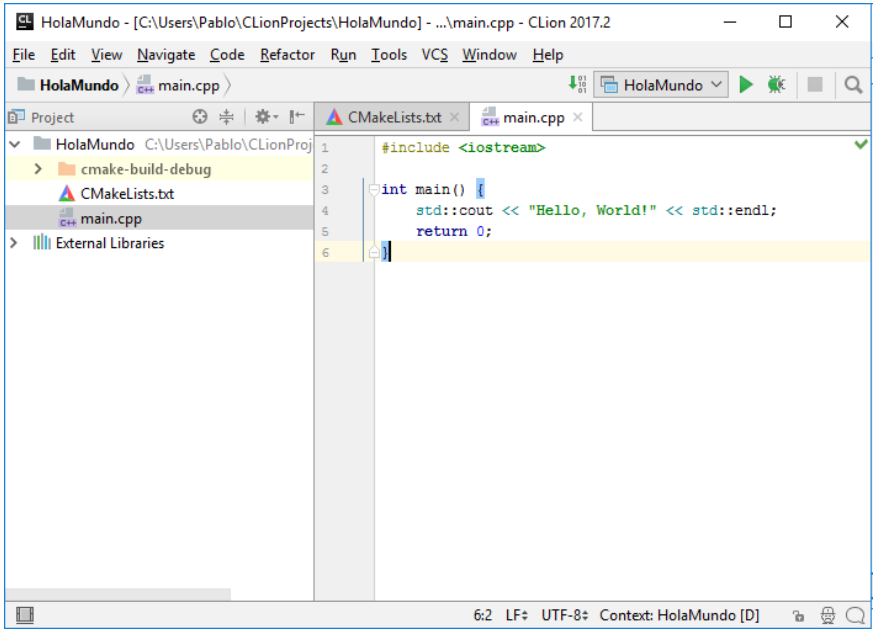


Proyecto Hola Mundo

- CLion genera autoáticamente dos archivos para el proyecto:
  1. main.cpp
  2. CMakeList.txt
- y muestra al usuario la IDE



Proyecto Hola Mundo



- Primera sorpresa: En el editor de texto, podemos ver que el archivo main.cpp no está vacio.
-

```
1  #include <iostream>
2
3  int main() {
4      std::cout << "Hello, World!" << std::endl;
5      return 0;
6  }
```

Proyecto Hola Mundo

```
1  #include <iostream>
2
3  int main() {
4      std::cout << "Hello, World!" << std::endl;
5      return 0;
6  }
```

¿Que hace el programa?

Proyecto Hola Mundo

```
1  #include <iostream>
2
3  int main() {
4      std::cout << "Hello, World!" << std::endl;
5      return 0;
6  }
```

- `#include <iostream>` sirve para incorporar al programa un set de funciones.
  1. "#" es un caracter que indica al compilador una instrucción especial
  2. "include" instrucción que incluye la librería que puede estar entre llaves o parentesis
  3. "iostream" es una libreria de funciones para el manejo de entrada/salida de c++

Proyecto Hola Mundo

```
1  #include <iostream>
2
3  int main() {
4      std::cout << "Hello, World!" << std::endl;
5      return 0;
6  }
```

- `int main() {` define el inicio del bloque de la función principal del programa. Por convención, el programa siempre inicia su ejecución en esta funcion. Podemos ver dos características:
  1. Esta función main no recibe parámetros "()"
  2. La función devuelve una variable de tipo **int** o sea entero.
  3. { Es la llave de apertura de la función.

Proyecto Hola Mundo

```
1  #include <iostream>
2
3  int main() {
4      std::cout << "Hello, World!" << std::endl;
5      return 0;
6  }
```

- `std::cout << "Hello, World!" << std::endl;`
  1. `std` es una librería de funciones definida dentro de iostream. Para acceder a una de esas funcions se usan dos puntos consecutivos `::`
  2. `cout` es una instrucción para imprimir en pantalla el mensaje entre << y <<, aqui es un mensaje de tipo String, entre comillas ", pero es capaz de imprimir todo tipo de variables.
  3. La instrucción `endl` indica el fin de la línea (retorno de carro)
  4. `;` finaliza la instrucción. El compilador lo precisa para identificar que termina la línea de instrucción. Olvidarse el punto y coma representa el 50 % de los errores de compilación en C++ :-)

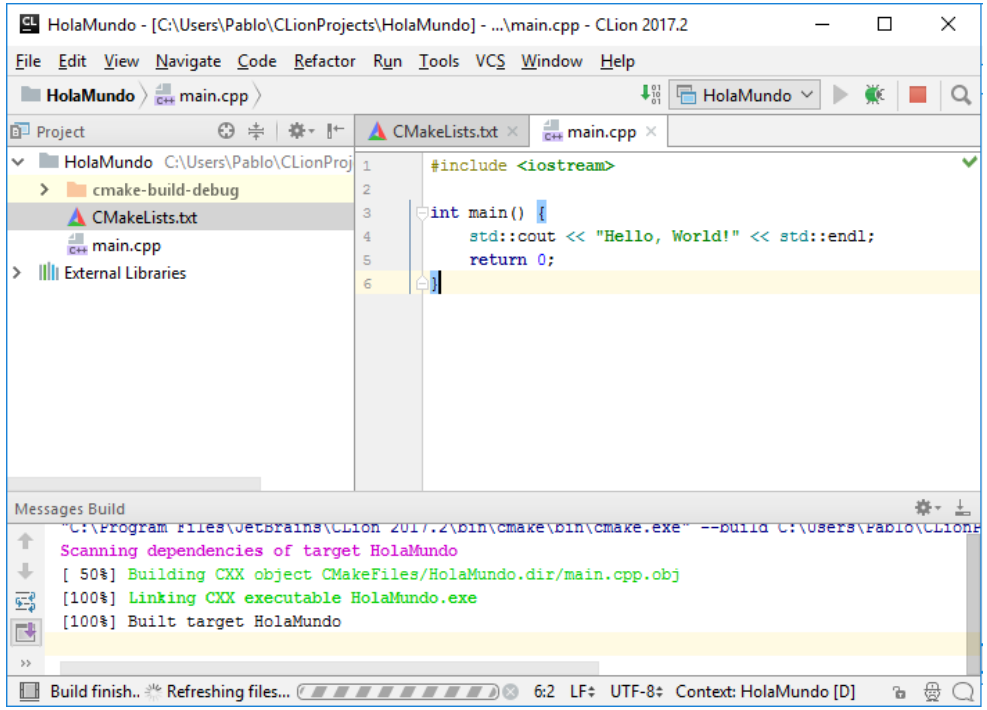
Proyecto Hola Mundo

```
1  #include <iostream>
2
3  int main() {
4      std::cout << "Hello, World!" << std::endl;
5      return 0;
6  }
```

- `return 0;` El programa termina retornando 0, que significa que no hubo errores en la ejecución.
- `}` La llave de cierre termina el bloque de la función main. Olvidarse de los cierres de bloques es otra fuente común de error de compilación.

Proyecto Hola Mundo

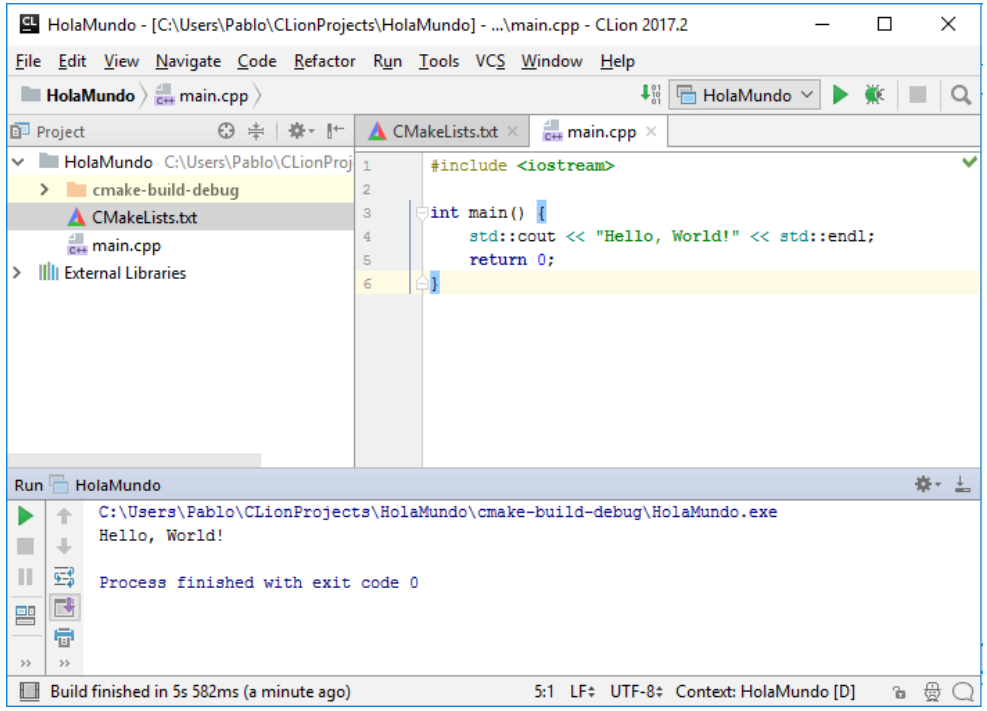
- Ahora vamos a compilar el programa para generar el archivo ejecutable.
- Esto se hace con CLion utilizando el icono verde de RUN.
- La IDE abre un nuevo panel que muestra el avance y estado de la compilación.



- Podemos ver que el cmake ejecutó secuencialmente una compilación, un linking y terminó por construir el ejecutable.

Proyecto Hola Mundo

- Luego de la compilación, el IDE ejecuta automáticamente el programa y muestra el resultado en un nuevo panel.



- En este panel podemos ver efectivamente, el mensaje generado por la función `cout`.

Proyecto Hola Mundo

- Que pasa cuando la compilación falla por un error?
- Ejecutemos el programa con errores de sintaxis:
  1. Quitar el punto y coma al final de la linea del `cout`.
  2. Borrar la llave de cierre del `main`.
- Analizar el mensaje de error que nos devuelve el IDE.
- Es útil para resolver el problema?.

Bibliografía

- B. Stroustrup. The C++ Programming Language.
  - Part I: Introductory Material - A Tour of C++: The Basics