

Introducción a la programación imperativa

Algoritmos y Estructuras de Datos I

Repaso: Teórica

- ▶ Lógica proposicional (binaria, trivaluada)
- ▶ Tipos de datos Básicos: enteros, reales, booleanos, char, enumerados
- ▶ Tipos de datos Compuestos: tuplas y secuencias
- ▶ Cuantificadores (\forall , \exists)
- ▶ predicados y funciones
- ▶ Especificación de problemas

Repaso: Tipos de datos

- ▶ 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.
- ▶ El lenguaje de especificación que presentamos tiene distintos tipos de datos:
 - ▶ enteros (\mathbb{Z})
 - ▶ reales (\mathbb{R})
 - ▶ valores de verdad (Bool)
 - ▶ caracteres (Char)
 - ▶ secuencias ($seq(T)$)

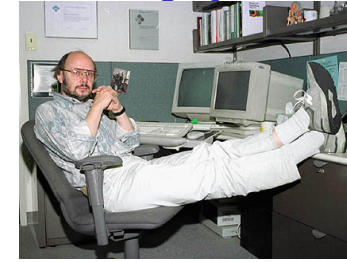
Paradigmas de lenguajes de programación

- ▶ **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

Programación imperativa

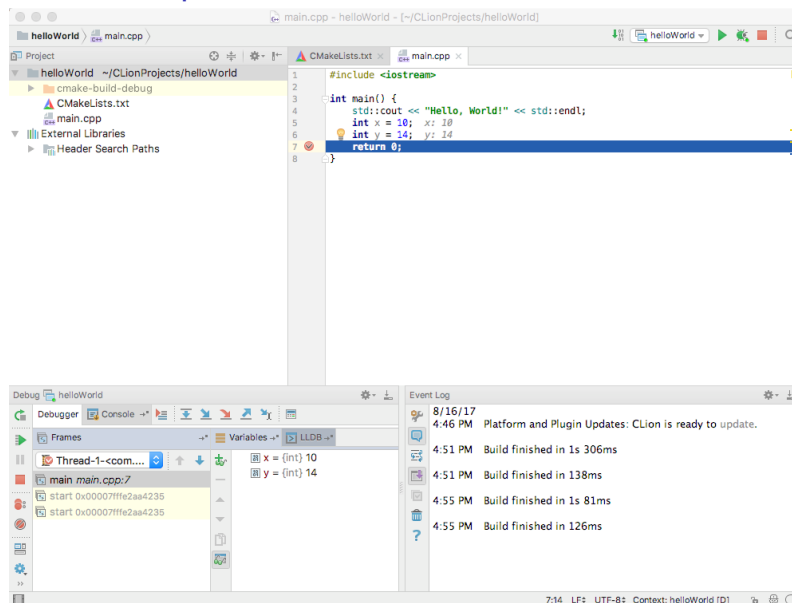
- **Entidad fundamental:** **variables**, que corresponden a posiciones de memoria (RAM) y cambian explícitamente de valor a lo largo de la ejecución de un programa.
 - ⇒ Pérdida de la transparencia referencial
- **Operación fundamental:** **asignación**, para cambiar el valor de una variable.
 1. Una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

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 → new C → C with Classes → **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**.

Repaso Taller: presentamos CLion



Repaso Taller: presentamos el lenguaje C/C++

- C y C++ son lenguajes **compilados**: Los archivos .c/.cpp con el **código fuente** son traducidos a **lenguaje de máquina** (en archivos .exe), que es ejecutable por el procesador.
 1. El lenguaje de máquina depende de la **plataforma** (hardware y sistema operativo).
 2. Se requiere un **compilador** para la plataforma en cuestión.
 3. El código fuente es el mismo, pero el resultado de la compilación es distinto para cada plataforma.
- Un programa en C es una colección de **funciones**, con una función principal llamada **main()**.

Demo #1: El primer programa escrito en C++

- Una versión minimal de “Hola, mundo” en C++:

```
1 #include <iostream>
2 using namespace std;
3
4 int main() {
5     cout << "Hola, mundo!" << endl;
6     return 0;
7 }
```

- ¿Qué pasa cuando lo compilamos y corremos en CLion?

Hola, mundo!

Process finished with exit code 0

- El exit code 0 indica que la ejecución finalizó normalmente.

Programación Imperativa: Variables

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

```
1 #include <iostream>
2 using namespace std;
3
4 int main() {
5     int a = 11;
6     cout << a;
7     return 0;
8 }
```

- A partir de la línea 4, 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 $\mathbb{Z} = \text{int}$ y $\mathbb{R} = \text{float}$.

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 = 5; // Declaracion + Inicializacion  
3     a = 7+2; // Asignacion  
4     ...  
5 }
```

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 using namespace std;  
4  
5 int main() {  
6     float x = 2 + 5;  
7     float y = sin(x) + cos(x);  
8  
9     cout << y;  
10    return 0;  
11 }
```

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 using namespace std;  
3  
4 int main() {  
5     cout << "Ingrese un valor entero" << endl;  
6     int valor = 0;  
7     cin >> valor;  
8     cout << "El valor ingresado fue " << valor << endl;  
9     return 0;  
10 }
```

Demo #3: Expresiones indefinidas en C++

¿Qué pasa al ejecutar este programa?

```
1 #include <iostream>
2 using namespace std;
3
4 int main() {
5     cout << "Ingrese un valor por teclado " << endl;
6     int x;
7     cin >> x;
8     int r = 1 / x;
9     cout << "Su inverso multiplicativo es " << r << endl;
10    return 0;
11 }
```

- ▶ ¿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
 - ▶ Continúa 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**.

Variables: Actualización

- ▶ Más aún, las variables pueden ir cambiando de valor a lo largo de la ejecución de un programa!

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

- ▶ Necesitamos mecanismos para analizar estos efectos!
- ▶ **Debugging**: Ejecución del programa **paso a paso** para analizar su comportamiento.

Demo #4: Debugging en CLion

```
1 #include <iostream>
2 using namespace std;
3
4 int main() {
5     int a=0;
6     int b=0;
7     cout << "Ingrese un nuevo valor para a " << endl;
8     cin >> a;
9     cout << "Ingrese un nuevo valor para 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 }
```

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 }
```

Demo #5: Debug de Llamados a funciones

Step over vs. Step into

```
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 << "valor calculado de b es " << 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 }
```

Implementación de especificaciones

Podemos usar C++ para escribir programas que implementan especificaciones. Por ejemplo, dada la siguiente especificación de la función inc:

```
proc inc(in a:ℤ, out r:ℤ) {
    Pre { True }
    Post { r = a + 1 }
}
```

La siguiente función de C++ implementa a inc:

```
1 int inc(int a) {
2     int r = a + 1;
3     return r;
4 }
```

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
- ▶ Declaración de nuevas funciones
- ▶ Llamados (invocaciones) a funciones (ya sea declaradas o en Bibliotecas)
- ▶ Ejecución secuencial de sentencias (ordenes/instrucciones)
- ▶ IDE: CLion

Intervalo

Break!

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

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

- ▶ **Ejemplo:** Sea la siguiente especificación para el problema de calcular el valor absoluto de un entero:

```
proc valorAbsoluto(in x :  $\mathbb{Z}$ , out result :  $\mathbb{Z}$  ){  
  Pre { True }  
  Post { result = | x | }  
}
```

- ▶ ¿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.

⇒ 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 también 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

```
► proc sumar(in n :  $\mathbb{Z}$ , out result :  $\mathbb{Z}$  ){  
    Pre {  $n \geq 0$  }  
    Post {  $result = \sum_{i=1}^n i$  }  
}
```

```
►  
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

- ▶ Sea la siguiente especificación del problema de decidir si un número $n \geq 2$ es primo.

```
proc primo(in n :  $\mathbb{Z}$ , out result : Bool ){
  Pre { $n \geq 2$ }
  Post {result = true  $\leftrightarrow$  esPrimo(n)}
}
```

Otro ejemplo usando “for”

Podemos escribir un programa que implementa a la especificación 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.

Bibliografía

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