Introducción a la programación imperativa

Algoritmos y Estructuras de Datos I

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.

  $\Rightarrow$ 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.

Lenguaje C

- El lenguaje C fue creado por Dennis Ritchie entre 1969 y 1973 en Bell Labs, para una reimplementación de Unix.
- Derivado del lenguaje no tipado "B" (D. Ritchie y K. Thompson, 1969).
- Etimología: Bon $\rightarrow$ B $\rightarrow$ New B $\rightarrow$ C.

Lenguaje C++

- El lenguaje C++: fue creado por Bjarne Stroustrup en 1983.
- Etimología: C → new C → C with Classes → C++.
- Lo usamos como lenguaje imperativo (también soporta parte del paradigma de objetos).

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**.

El primer programa

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

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

Variables

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

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

- 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 C++

- Recordemos que 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 usa distintos tipos de datos:
  - enteros ($\mathbb{Z}$)
  - reales ($\mathbb{R}$)
  - valores de verdad (Bool)
  - caracteres (Char)
  - secuencias

Tipos de datos C++ vs. especificación

- En C++ tenemos tipos de datos que implementan parcialmente cada uno de estos tipos:
  - El tipo `int` para números enteros
  - El tipo `float` para números reales
  - El tipo `bool` para valores booleanos
  - El tipo `char` para caracteres
- Ni `int` ni `float` contienen todos los valores de $\mathbb{Z}$ y $\mathbb{R}$, pero a los fines de AED1, vamos a asumir que $\mathbb{Z} = \texttt{int}$ y $\mathbb{R} = \texttt{float}$.

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  {
3    int a = "Hey, hey!"; // No! La expresion asignada no es un int
4    ...
5  }
```

Declaración y asignación

- 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  {
3    int a = 5; // Declaracion + Inicializacion
4    a = 7+2; // Asignacion
5    ...
6  }
```

Expresiones y funciones

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

```
1  #include <iostream>
2
3  int main()
4  {
5    float x = 2 + 5;
6    float y = sin(x) + cos(x);
7
8    std::cout << y;
9    return 0;
10 }
```

Funciones con valores de retorno

- Podemos declarar nuestras propias funciones. Para eso debemos especificar:
  1. Tipo de retorno
  2. Nombre
  3. Argumentos (o parámetros)
- 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

- La función devuelve el valor que se especifica mediante la sentencia **return**.
- Por ejemplo, la siguiente función toma una parámetro entero y devuelve el siguiente valor:

```
1  int siguiente(int a)
2  {
3    return a+1;
4  }
```

- Otra versión, con código poco estándar:

```
1  int siguiente(int a)
2  {
3    int b;
4    b = a+1;
5    return b;
6  }
```

Funciones con valores de retorno

- Volviendo al ejemplo anterior, podemos llamar a **siguiente()** dentro de nuestro código:

```
1  #include <iostream>
2
3  int siguiente(int a)
4  {
5    return a+1;
6  }
7
8  int main()
9  {
10   int a = 5;
11   int b = 2 * siguiente(a);
12
13   std::cout << b;
14   return 0;
15 }
```

Expresiones y funciones

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

```
1  int suma(int a, int b)
2  {
3    return a+b;
4  }
5
6  int main()
7  {
8    int a = suma(2,3);
9    std::cout << a;
10   return 0;
11 }
```

Expresiones y funciones

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

```
1  int cuentas(int a, int b, int c)
2  {
3    int d = 2*a + b;
4    int e = 3*d - (c/a);
5    int f = e + 2*d;
6    return f;
7  }
```

- La ejecución de un programa imperativo tiene temporalidad.

Expresiones y funciones

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

```
1  int cuentas(int a, int b, int c)
2  {
3    int d = 2*a + b;
4    int e = d + a;
5
6    b = d + c/2;
7    a = a + 2; // (!)
8
9    return e + b/a;
10 }
```

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

Entornos de desarrollo

- Existen diversos compiladores y entornos de desarrollo para C/C++.
  1. CodeBlocks
  2. djgpp
  3. eclipse
  4. ...
- Veamos a continuación el entorno de desarrollo eclipse! Aprovechemos también para ver en funcionamiento los programas (debugging).

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. 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
```

- 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:** Calcular el valor absoluto de un entero:

```
1  int abs(int n)
2  {
3    int res;
4
5    if( n > 0 )
6      res = n;
7    else
8      res = -n;
9
10   return res;
11 }
```

- Luego de la instrucción alternativa, la variable **res** contiene el valor buscado.

Instrucción alternativa

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

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

- Se debe tener cuidado con estas construcciones, porque **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.

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

- 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 $n \neq 0$.

Recursión en C/C++

- Podemos hacer funciones recursivas 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  {
3    if( n == 0 )
4      return 0;
5    else
6      return n + suma(n-1);
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.

Ciclos

- 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.

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  {
3    int i = 1;
4    int sum = 0;
5
6    while( i <= n )
7    {
8      sum = sum + i;
9      i = i + 1;
10   }
11
12   return sum;
13 }
```

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)
3    sum = sum + i;
```

Otro ejemplo

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

```
1  boolean primo(int n)
2  {
3    int divisores = 0;
4    for(int i=2; i<n; ++i)
5    {
6      if( n % i == 0 )
7        divisores += 1;
8    }
9
10   return divisores == 0;
11 }
```