Capítulo

02

Conceptos Básicos
de Programación

Ania Cravero Leal

Mauricio Diéguez Rebolledo

Departamento de Ingeniería de Sistemas
Facultad de Ingeniería, Ciencias y Administración

"Proyecto financiado por el Fondo de Desarrollo Educativo de la Facultad de Ingeniería, Ciencias y Administración de la Universidad de La Frontera"

Versión

0.9

TEMARIO

Conceptos Básicos de **Programación**

En este segundo capítulo describiremos una serie de conceptos básicos que nos ayudará a comprender la estructura de un programa.

Comenzaremos con una descripción general de un programa, los lenguajes de programación y su clasificación. A continuación introduciremos los conceptos básicos que se deben ser comprendidos: *datos, variables, constantes, tipos de datos, variables y operadores*; para luego describir el proceso de creación de programas mediante una serie de herramientas de edición, compilación y ejecución. Finalmente, describiremos de manera breve el Ciclo de Vida del Software de manera de comprender de una manera simple las etapas de análisis, diseño, implementación y mantención de un programa.

2.1 Programas y Lenguajes de Programación

En el capítulo 1 ya presentamos una pequeña definición de programa. De manera descriptiva, podemos decir que un *programa* es un texto, una colección de líneas que se escriben en algún

Ejemplo 2.1:
Ejemplo de un programa escrito en lenguaje de programación C++ y Java.

HolaMundo.cpp

```
#include <iostream>
using  namespace std;

int main()
{ cout << "Hola Mundo…" << endl;
  cout << "Hoy es Lunes…" << endl;
return 0;
}
```

HolaMundo.java

```
import java.io.*;

public class HolaMundo {
  public static void main(String[] args)
{
System.out.println ("Hola Mundo");
  }
}
```

lenguaje de programación empleando una herramienta de edición. Ese texto se llama *código fuente*, y no puede ser (en general) ejecutado por el computador ya que está escrito en un lenguaje de programación de alto nivel. A continuación aclaramos estos conceptos.

a. Programa:

Es una colección de instrucciones que, debidamente traducidas, serán ejecutadas por un computador para realizar una tarea. En otras palabras, un programa es la descripción de un algoritmo escrita en cierto lenguaje de programación. (Ver también en el capítulo 1, sección 1.1.2 b.)

b. Algoritmo:

De manera sencilla, podemos decir que un algoritmo es un método que se emplea para obtener resultados a partir de los datos disponibles. En el capítulo xx se presenta con mayor detalle este concepto.

c. Lenguaje de Programación:

Es una colección de reglas que especifican la forma en que deben escribirse sentencias o instrucciones para que sea posible traducirlas a otras comprensibles para el procesador en que se ejecutarán. (ver también en el capítulo 1, sección 1.1.2 d.)

Los programas o código fuente se escriben mediante un editor en un archivo de texto y quedan almacenados en alguna memoria secundaria, por ejemplo, el archivo HolaMundo.cpp contiene un ejemplo de programa escrito en lenguaje de programación C++, y HolaMundo.java contiene un ejemplo de programa escrito en lenguaje de programación Java.

2.2 Clasificación de los lenguajes de programación.

Los lenguajes de programación se pueden clasificar según varios criterios. Hemos encontrado **doce** en total: Nivel de abstracción, propósito, evolución histórica, manera de ejecutarse, manera de abordar la tarea a realizar, paradigma de programación, lugar de ejecución, concurrencia, interactividad, realización visual, determinismo y productividad. Describiremos sólo algunos de ellos. (Guevara, 2008).

2.2.1 Nivel de Abstracción

Según el nivel de abstracción, es decir, según el grado de cercanía a la máquina:

- ***Lenguaje de bajo nivel:*** Son lenguajes totalmente dependientes de la máquina, es decir que el programa que se realiza con este tipo de lenguaje no se puede migrar o utilizar en otras máquinas. Al estar prácticamente diseñados a medida del hardware, aprovechan al máximo las características del mismo. La programación se realiza teniendo muy en cuenta las características del procesador. Normalmente se utiliza una secuencia de bit, es decir, de ceros y unos.

- ***Lenguaje de nivel medio:*** Permiten un mayor grado de abstracción pero al mismo tiempo mantienen algunas características de los lenguajes de bajo nivel. El lenguaje ensamblador es un derivado del lenguaje de máquina y está formado por abreviaturas de letras y números. Con la aparición de este lenguaje se crearon los programas traductores para poder pasar los programas escritos en lenguaje ensamblador a lenguaje de máquina. Como ventaja con respecto al lenguaje de máquina es que los códigos fuentes eran más cortos y los programas creados ocupaban menos memoria. Nota: C puede realizar operaciones lógicas y de desplazamiento con bits, tratar todos los tipos de datos como los que son en realidad a bajo nivel (números), etc.

- ***Lenguaje de alto nivel:*** Son lenguajes más parecidos al lenguaje humano. Utilizan conceptos, tipos de datos, etc., de una manera cercana al pensamiento humano ignorando (o abstrayéndose) del funcionamiento de la máquina. Se tratan de lenguajes independientes de la arquitectura del computador. Por lo que, en principio, un programa escrito en un lenguaje de alto nivel, se puede migrar de una máquina a otra sin ningún problema. Estos lenguajes permiten al programador olvidarse por completo del funcionamiento interno de la máquina para la que están diseñando el programa. Tan sólo se necesita un traductor que entienda el

código fuente como las características de la máquina. Ejemplos: Java, Ruby, C, C++, Cobol, PHP, entre otros.

2.2.2 Propósito

Según el propósito, es decir, el tipo de problemas a tratar con ellos:

- ***Lenguaje de propósito general:*** Son lenguajes aptos para todo tipo de tareas: Ejemplo: C, C++, Java, entre otros.
- ***Lenguaje de propósito específico:*** Hechos para un objetivo muy concreto. Ejemplo: Csound (para crear archivos de audio).
- ***Lenguaje de programación de sistemas:*** Diseñados para crear sistemas operativos o drivers. Ejemplo: C.
- ***Lenguaje script:*** Para realizar tareas de control y auxiliares. Antiguamente eran los llamados lenguajes de procesamiento por lotes (batch) o JCL ("Job Control Languages"). Se subdividen en varias clases (de shell, de GUI, de programación web, etc.). Ejemplos: bash (shell), mIRC script, JavaScript (programación web).

2.2.3 Evolución histórica

Con el paso del tiempo, se va incrementando el **nivel de abstracción**, pero en la práctica, los lenguajes de una generación no terminan de sustituir a los de la anterior:

- ***Lenguaje de primera generación (1GL):*** Son los lenguajes que permitían crear programas mediante el código de máquina.
- ***Lenguaje de segunda generación (2GL):*** Son aquellos lenguajes de programación del tipo ensamblador. Este tipo de lenguaje permite algunas instrucciones como de suma, de salto, y para almacenar en memoria.
- ***Lenguaje de tercera generación (3GL):*** Estos son la mayoría de los lenguajes modernos, diseñados para facilitar la programación a los humanos. Ejemplos: C, Java, C++, entre otros.

- ***Lenguaje de cuarta generación (4GL):*** Diseñados con un propósito concreto, es decir, para abordar un tipo concreto de problemas. Ejemplos: NATURAL, Mathematica.

- ***Lenguaje de quinta generación (5GL):*** La intención es que el programador establezca el "qué problema ha de ser resuelto" y las condiciones a reunir, y la máquina lo resuelve. Son utilizados en inteligencia artificial. Ejemplo: Prolog.

2.2.4 Manera de ejecutarse

Según la manera de ejecutarse:

- ***Lenguaje compilado:*** Un programa traductor traduce el código del programa (código fuente) en código máquina (código objeto). Otro programa, el enlazador, unirá los archivos de código objeto del programa principal con los de las librerías para producir el programa ejecutable. Ejemplo: C.

- ***Lenguaje interpretado:*** Un programa (intérprete), ejecuta las instrucciones del programa de manera directa. Ejemplo: Lisp.

También los hay mixtos, como Java, que primero pasan por una etapa de compilación en la que el código fuente se transforma en "bytecode", y este "bytecode" puede ser ejecutado luego (interpretado) en computadores con distintas arquitecturas (procesadores) que tengan todos instalados la misma "máquina virtual" Java.

2.2.5 Manera de abordar la tarea a realizar

Según la manera de abordar la tarea a realizar, pueden ser:

- **Lenguaje imperativo**: Indican cómo hay que hacer la tarea, es decir, expresan los pasos a realizar. En este caso, cada instrucción del programa es como una orden que se le entrega al computador para que las ejecute. Ejemplo: C.

- **Lenguaje declarativo**: Estos lenguajes, indican qué hay que hacer. En este caso, cada instrucción es una acción a ejecutar en el computador. Ejemplos: Lisp, Prolog. Otros ejemplos

de lenguajes declarativos, pero que no son lenguajes de programación, son HTML (para describir páginas web) o SQL (para consultar bases de datos).

2.2.6 Paradigma de programación

El **paradigma de programación** es el estilo de programación empleado. Algunos lenguajes soportan varios paradigmas, y otros sólo uno. Se puede decir que históricamente han ido apareciendo para facilitar la tarea de programar según el tipo de problema a abordar, o para facilitar el mantenimiento del software, o por otra situación similar, por lo que todos corresponden a lenguajes de alto nivel (o nivel medio), estando los lenguajes ensambladores "atados" a la arquitectura de su procesador correspondiente. Los principales son:

- ***Lenguaje de programación procedural:*** Este tipo de lenguaje divide el problema en partes más pequeñas, que serán realizadas por subprogramas (subrutinas, funciones, procedimientos), que se llaman unas a otras para ser ejecutadas. Ejemplos: C, Pascal, C++.

- ***Lenguaje de programación orientada a objetos:*** Crean un sistema de clases y objetos siguiendo el ejemplo del mundo real, en el que unos objetos realizan acciones y se comunican con otros objetos. Ejemplos: C++, Java.

- ***Lenguaje de programación funcional:*** La tarea se realiza evaluando funciones, (como en Matemáticas), de manera recursiva. Ejemplo: Lisp.

- ***Lenguaje de programación lógica:*** La tarea a realizar se expresa empleando lógica formal matemática. Estos lenguajes expresan qué computar. Ejemplo: Prolog.

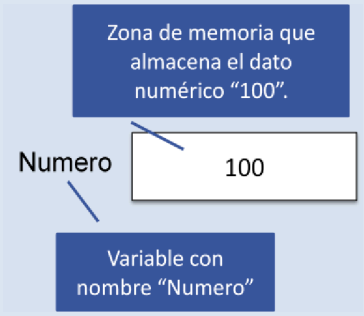
Hay muchos paradigmas de programación: Programación genérica, programación reflexiva, programación orientada a procesos, entre otros.

Preguntas para el Lector

i. ¿Conoces algún lenguaje de programación? Si es así, ¿de qué manera lo clasificarías?
ii. De acuerdo a la clasificación que leíste. ¿Cuáles crees tú que son los lenguajes apropiados que deben aprender en la carrera que estudias?

Ejemplo 2.2:

Este ejemplo muestra una variable (zona de memoria) cuyo nombre es "Numero", y almacena el dato "100" que es un número de tipo entero.



OJO:

Nombres de variables: Deben ser nombres que tengan relación con el dato que se almacena. Se puede utilizar caracteres y números pero no símbolos ni tildes. El único carácter especial que se permite es el guión bajo (_)

Ejemplo:
numero1
total
sumaTotal
nombre_per
auxiliar

2.3 Conceptos Básicos de Programación

Comenzamos este capítulo definiendo el término ***algoritmo***. Para comprender este término, necesitarás comprender algunos conceptos como:

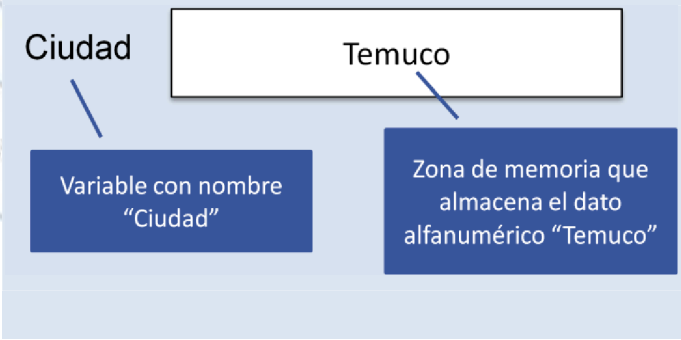
a. Dato o valor:

Unidad mínima de información sin sentido en sí misma, pero que adquiere significado en conjunción con otras precedentes de la aplicación que las creó. Un ejemplo de dato es un número cualquiera que no representa nada en particular, o una letra, un símbolo, etc.

b. Variable:

En programación, una variable es un espacio de memoria reservado para almacenar un ***valor*** que corresponde a un ***tipo de dato*** soportado por el lenguaje de programación. Una variable es representada y usada a través de una etiqueta (un nombre) que le asigna un programador o que ya viene predefinida.

Ejemplo 2.3:
La siguiente variable, con nombre "ciudad", almacena un dato de tipo alfanumérico, "Temuco"



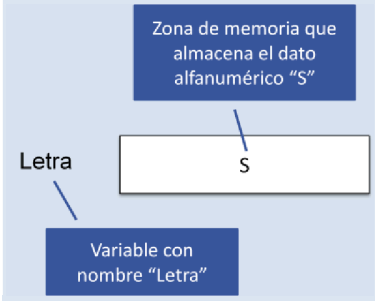
Ejemplo 2.4:
El número 145 es un tipo de dato numérico

Ejemplo 2.5:
m es un tipo de dato alfanumérico

Ejemplo 2.6:
1 puede ser considerado un tipo de dato entero, o también un tipo de dato booleano, ya que 1 significa verdadero (true) y 0 (cero) significa falso (false) para el computador.

Ejemplo 2.7:
16896.98 es un tipo de dato numérico

Ejemplo 2.8:
La variable ***letra*** almacena un dato de tipo alfanumérico (o conocido como caracter). Por lo tanto es una variable de tipo alfanumérico.



c. Tipo de dato:

Es una restricción impuesta para la interpretación, manipulación, o representación de datos. Tipos de datos comunes en lenguajes de programación son los tipos primitivos, ***numéricos, caracteres o alfanuméricos, y booleanos***.

Tipo alfanumérico: Estos representan todos los caracteres del código ASCII, que incluye las letras desde la a – z, A – Z, números de 0 a 9, y símbolos (incluye los tildes, la ñ y espacio)

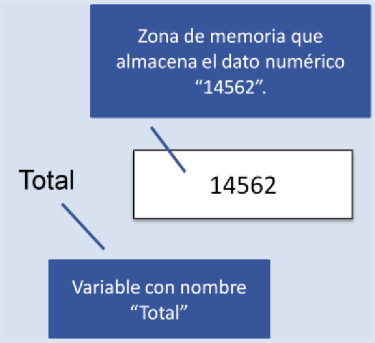
Aclaración: Si 1234 es almacenado en una variable de tipo numérico, es posible realizar operaciones **matemáticas con él. En este caso 1234 + 1 es 1235, pues** el computador los ha sumado.

En cambio si 1234 es almacenado en una variable que almacena datos alfanuméricos, entonces 1234 + 1 es 12341, pues el computador lo ha concatenado.

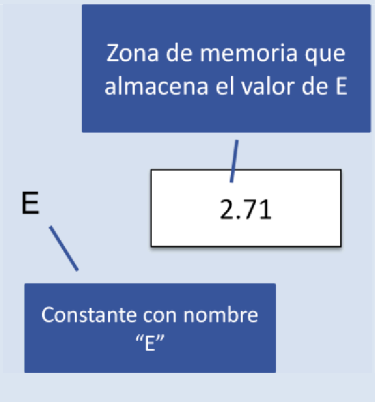
d. Tipo de variable:

Una variable puede ser del tipo booleano, entero, decimal de coma flotante, alfanumérico, etc. El tipo que se le asigna a un variable tiene que ver con el tipo de dato que almacena su espacio de memoria. Por lo tanto, la variable "Ciudad", del ejemplo 2.3, es de tipo alfanumérico, y decimos que es una variable alfanumérica.

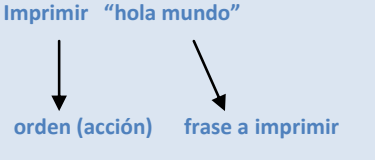
Ejemplo 2.9:
La variable ***Total*** almacena un dato de tipo numérico (entero). Por lo tanto es una variable de tipo entero.



Ejemplo 2.10:
La siguiente constante almacena el valor de **PI**. Y es un dato que nunca cambia.



Ejemplo 2.12:
La siguiente instrucción le da una orden al computador, la orden es imprimir en la pantalla una frase



e. Constante:

Una constante es una zona de memoria que almacena un dato al igual que una variable. Lo diferente, es que el dato almacenado no puede cambiar.

Preguntas para el lector:

i. ¿Tu edad de qué tipo de dato es?

ii. ¿Tu nombre de qué tipo de dato es?

iii. ¿Puedes clasificar los tipos de datos que se pueden almacenar en una zona de memoria?

iv. ¿Cuál es la diferencia principal entre los conceptos de constante y variable?

f. Instrucción:

Una instrucción es una única operación de un computador. Es como darle una orden al procesador. En general una instrucción se puede componer de ***datos, variables, constantes y operadores***.

Con estas aclaraciones podemos comprender de mejor manera qué es un algoritmo, que en palabras simples seria:

Algoritmo: ……."*un conjunto de órdenes o instrucciones que se ejecutan en forma secuencial, procesando distintos datos para dar solución a un problema general*"……

2.4 Operadores

2.4.1 Operadores Aritméticos

La tabla 2.1 describe los operadores aritméticos que permiten resolver operaciones matemáticas básicas.

También se puede utilizar el uso de paréntesis en las fórmulas matemáticas para cuestiones de precedencia.

Tabla 2.1: Operadores aritméticos

Operador	Descripción	Símbolo	Ejemplo
Suma	Permite sumar dos datos	+	a + b
Resta	Permite restar dos datos	-	a - b
Multiplicación	Permite multiplicar dos datos	*	a * b
División	Permite dividir dos datos. Se debe considerar que la división por cero causa un error.	/	a / b
Módulo	Entrega el resto de una división entera	%	a % b

2.4.2 Operadores Relacionales

Los operadores relacionales son símbolos que se usan para comparar dos valores. Si el resultado de la comparación es correcto la expresión considerada es verdadera, en caso contrario es falsa. Por ejemplo, 8>4 (ocho mayor que cuatro) es verdadera, se representa por el valor **true** del tipo básico **boolean**, en cambio, 8<4 (ocho menor que cuatro) es falsa, **false**.

La tabla 2.2 describe los operadores relacionales que son utilizados por los lenguajes de programación.

Tabla 2.2: Operadores Relacionales

Operador	Descripción	Símbolo	Ejemplo
Igual	Permite comparar si dos datos son iguales. Entrega verdadero si lo son	= =	a == b
Menor que	Permite comparar si el primer dato es menor que el segundo.	<	a < b
Menor o igual que	Permite comparar si el primer dato es menor o igual que el segundo.	<=	a <= b
Mayor que	Permite comparar si el primer dato es mayor que el segundo.	>	a > b
Mayor o igual que	Permite comparar si el primer dato es mayor o igual que el segundo.	>=	a >= b
Distinto	Permite comparar si dos datos son distintos. Entrega verdadero si lo son	≠ ó !=	a != b

2.4.3 Operadores Lógicos

Los operadores lógicos son:

- AND (el resultado es verdadero si ambas expresiones son verdaderas)
- OR (el resultado es verdadero si alguna expresión es verdadera)
- NOT (el resultado invierte la condición de la expresión)

AND y OR trabajan con dos operandos y entregan un valor lógico que está basado en las denominadas tablas de verdad. El operador NOT actúa sobre un operando. Estas tablas de verdad son conocidas y usadas en el contexto de la vida diaria, por ejemplo: "si hace sol Y tengo tiempo, iré a la playa", "si NO hace sol, me quedaré en casa", "si llueve O hay viento, iré al cine". Las tablas de verdad de los operadores AND, OR y NOT se muestran en las tablas siguientes:

Tabla 2.3: El operador lógico AND

x	y	resultado
true	true	true
true	false	false
false	true	false
false	false	false

Tabla 2.4: El operador lógico NOT

x	resultado
true	false
false	true

Ejemplo 2.13:
A la variable numero se le asigna el dato 21
numero ← 21

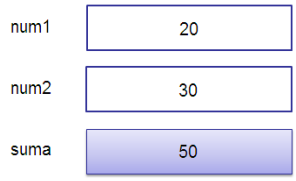
Ejemplo 2.14:
A la variable nombre se le asigna una palabra o una cadena de caracteres
nombre ← "Jorge"

Cabe notar que Jorge es una cadena de caracteres y no una variable. Esto lo podemos notar por la inclusión de comillas. En caso contrario, Jorge sería una variable a la que no se le ha asignado un dato.

Ejemplo 2.15:
A la variable total_num se le asigna la suma de dos números. Por lo tanto a la variable se le asigna el dato 50
total_num ← 16 + 34

Ejemplo 2.16:
A la variable suma se le asigna la suma de dos variables. Como las variables num1 y num2 almacenan un dato numérico, entonces es posible sumar ambas variables y almacenar el resultado en la variable suma.
num1 ← 20
num2 ← 30
suma ← num1 + num2

Observa lo que ocurre con cada zona de memória.



Cabe destacar que es necesario que las variables num1 y num2 deban tener un dato almacenado, para que el computador pueda sumar el contenido de ambas y finalmente asignarlo en la variable suma.

Tabla 2.5: El operador lógico OR

x	y	resultado
true	true	true
true	false	true
false	true	true
false	false	false

Los operadores AND y OR combinan expresiones relacionales cuyo resultado viene dado por la última columna de sus tablas de verdad. Por ejemplo: (a<b) AND (b<c) es verdadero (**true**), si ambas son verdaderas. Si alguna o ambas son falsas el resultado es falso (**false**). En cambio, la expresión (a<b) OR (b<c) es verdadera si una de las dos comparaciones lo es. Si ambas, son falsas, el resultado es falso.

La expresión " NO *a* es menor que *b*" NOT (a<b) es falsa si *(a<b)* es verdadero, y es verdadera si la comparación es falsa. Por tanto, el operador NOT actuando sobre *(a<b)* es equivalente a (a>=b). La expresión "NO *a* es igual a *b*" NOT (a==b) es verdadera si *a* es distinto de *b*, y es falsa si *a* es igual a *b*.

2.5 La asignación

La instrucción fundamental en todo lenguaje de programación es ***la asignación***, que consiste en asociar un valor o dato a una variable. Todo algoritmo puede contemplarse como una combinación más o menos compleja de asignaciones. Su sintaxis es la siguiente:

X ← E

que se lee: "X toma por valor lo que valga E", o bien: "a X se le almacena el dato E". Dicha E debe ser una expresión válida del mismo tipo que X y que se pueda evaluar sin error en el momento de ejecutar la instrucción. Hay que considerar que primero se evalúa la expresión y a continuación se copia en la variable el dato obtenido, por lo tanto, este tipo de instrucciones debe leerse desde la derecha del símbolo (←) a la izquierda. Observa los ejemplos 2.13 al 2.18.

Ejemplo 2.17:
A la variable descuento se le asigna un dato no válido. Por tanto, es causa de error.
descuento ← 10 – des

En este caso la variable **des** no contiene ningún dato y por tanto el computador no podrá realizar la resta. Lo correcto sería hacer lo siguiente:

des ← 34
descuento ← 10 – des

Ejemplo 2.18:
A la variable resultado se le asigna el dato que obtenemos como resultado de la expresión matemática que está a la derecha del símbolo asignación.
……
resultado ← (a % 4) / (num1 – num2*3)

Recuerde que las variables **a, num1** y **num2** deben contener datos almacenados para que la asignación sea efectiva. De no ser así, la instrucción causará un error.

Ejemplo 2.19:
Ejemplo de un programa escrito en lenguaje de programación Java.

HolaMundo.java

```
public class HolaMundo{
  public static void main (String[ ]
args) {
System.out.printf("Hola
Mundo..%n);
Systema.out.printf("Hoy es
Lunes…");
  }
}
```

2.6 Cómo es realmente un Programa

Consideremos el ejemplo 2.19 del programa HolaMundo escrito en un lenguaje de programación Java:

Los programas, como hemos dicho, se escriben en archivos con formato de texto, solo que la extensión es .c, o .cpp, o .java, según corresponda el lenguaje de programación empleado.

Aún hay más, estos programas no se pueden utilizar directamente. El computador no comprende estas expresiones, aunque están escritas en un lenguaje de "alto nivel", lo cual significa que es comprensible por operadores humanos. Esto ocurre ya que el computador tiene otra visión del mundo (García-Bermejo, 2008).

En este sentido, podemos decir que no se puede ejecutar directamente un código fuente en un computador, ya que estas entienden únicamente instrucciones que en nada se parecen a las que se emplean en estos lenguajes de programación.

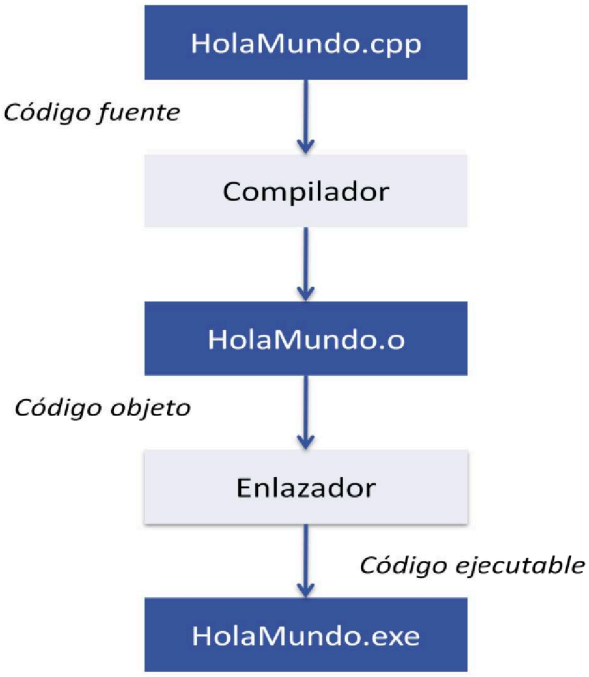
Por otro lado, las instrucciones aceptables para el computador (lenguaje de máquina) no son fácilmente interpretables por el humano. Así que tenemos una situación en que aparecen dos lenguajes: el lenguaje de programación de alto nivel, en que escribiremos nosotros el programa, y el lenguaje de máquina, propio de la máquina. Obviamente, hace falta traducir el texto creado por nosotros a instrucciones en código de máquina. Esta tarea la realizará un programa denominado *traductor* (ver definición en el capítulo 1, sección 1.1.2 e.).

Por tanto, para escribir un programa necesitamos al menos dos herramientas:

- **Un editor de texto**, para escribir un programa
- **Un traductor y enlazador**, para realizar la traducción de lenguaje de alto nivel a bajo nivel.

El editor de texto permite que el operador humano (es decir nosotros) escriba las instrucciones (en código fuente) que resuelven un determinado problema mediante un lenguaje de programación (ejemplo, el programa HolaMundo.cpp). Y el traductor y enlazador se encargarán de generar el código ejecutable en lenguaje de máquina (que sí es comprensible por el computador). La Figura 2.1 muestra el proceso.

Figura 2.1: El proceso de traducción



El proceso de traducción, mediante el cual se pasa de un archivo de texto (código fuente) a un archivo ejecutable, puede realizarse utilizando programas basados en línea de comando, y también programas basados en una interfaz gráfica de usuario o IDE (Integrated Development Environment o Entorno integrado de desarrollo), tema que se revisará en el capítulo 4.

Preguntas para el lector:

i. ¿En lenguaje de programación que utilizamos para resolver un determinado problema, lo comprende un computador? Explique
ii. ¿El lenguaje que utiliza el computador para ejecutar las instrucciones (órdenes que se le envían), es comprensible por el programador (nosotros)? Explique
iii. ¿Qué necesitamos para resolver este problema?

2.7 Buenas prácticas de programación

Nuestro enfoque en este libro está basado en los principios fundamentales que se aplican en todos los niveles de la computación: ***simplicidad***, que mantiene los programas breves y manejables; ***claridad***, que garantiza que sean fáciles de entender, tanto para las personas como para las máquinas; ***generalidad***, que significa que trabajarán bien en una amplia gama de situaciones y se adaptarán bien a medida que surjan situaciones nuevas; y ***automatización***, que permita que sea la máquina la que haga el trabajo por nosotros.

Deseamos compartir lecciones sobre asuntos prácticos, pasando el conocimiento a partir de nuestra experiencia y sugerir formas en que los programadores puedan ser más eficaces y productivos.

2.7.1 Estilo

Escribir un programa es más que tener buena sintaxis, corregir los errores y volverlo rápido. Los programas son leídos no sólo por los computadores sino también por los programadores. Un programa bien escrito es más fácil de entender y modificar que uno mal escrito. La disciplina de la buena escritura produce código con más probabilidades de ser correcto (Kernighan y Pike, 2000).

Los principios del estilo de la programación están basados en el sentido común guiado por la experiencia, no en reglas y recetas arbitrarias. El código debe ser claro y sencillo, aplicando una expresión natural, un lenguaje convencional, el uso de nombres con significado, el uso de formato limpio, y el uso de cometarios útiles. Seguiremos el estilo empleado en la libro “La práctica de la programación” de Kernighan y Pike, con pequeños ajustes de acuerdo al estilo propio de los académicos que participaron en la elaboración de este libro.

a. Nombres

Un nombre de variable o de función etiqueta un objeto y contiene información sobre su propósito. Debe ser informativo, conciso, fácil de recordar y, si es posible, fácil de pronunciar. Hay mucha información que

Ejemplo 2.20:
Nombres de variables adecuadas
n, nPuntos, contador, cont, suma, VentaTotal, anio, antigEmp, sueldoBase, areaAchu, areaCir, cont1, num2, numElement.
También existen algunas reglas para los nombres de variables dependiendo del lenguaje de programación a utilizar, como:
- el primer carácter siempre debe ser una letra
- se permite combinar letras y números
- no se puede utilizar espacios
- se puede utilizar el guión bajo "_"
Así que n_puntos, venta_total, también son nombres de variables adecuadas.

proviene del contexto y de alcance sintáctico; cuanto más amplio sea el alcance de una variable, mayor información deberá contener su nombre.

Las variables locales (dentro de una función) pueden tener **nombres cortos**. En este sentido, n puede ser suficiente, nPuntos es adecuado, y numeroDePuntos podría ser demasiado largo. Todo dependerá del contexto en donde se desarrolle el programa.

Preferentemente, los nombres de las variables se iniciarán con una letra en minúscula; y si necesitamos componerla con dos palabras, utilizaremos la mayúscula para iniciar la segunda palabra o bien el guión bajo (_). Veamos algunos ejemplos (2.10 a 2.22) de nombres de variables que podemos utilizar.

A veces se pide a los programadores emplear nom-bres largos para las variables, independientemente del contex-to, lo cual es un error: la claridad se logra muchas veces mediante la brevedad.

Para el caso de las constantes preferiremos el uso de mayús-culas. Esto nos permitirá diferen-ciar una constante de una variable.

Sobre la composición de los nombres en sí, emplear npendientes, o numPend, o num_pend es cuestión de gusto personal. En nuestro caso daremos preferencia a la forma numPend sólo con el fin de lograr un estilo que represente a los cursos de programación de la Universidad.

Ejemplo 2.21:
Nombres para constantes
E, PI, IVA, DDS, INICIO, FINAL, NUMELEMEN, TAMMAX
Otra recomendación es **utilizar nombres de variables que sean consistentes**. "Dé a las cosas relacionadas nombres que muestren esta relación y marquen sus diferencias".

b. Expresiones e instrucciones

Por analogía con la selección de nombres que ayuden a la comprensión del lector del programa, debemos escribir expresiones e instrucciones de forma tal que vuelvan su significad lo más transparente posible. En este sentido, debemos escribir el código fuente lo más claro posible con el fin de que cumpla su función.

Se recomienda el uso de espacios antes y después de los operadores para sugerir agrupaciones; más genéricamente, debemos aplicar un formato que ayude a la legibilidad. Esto es trivial, per valioso, similar a mantener su dormitorio ordenado para poder encontrar las cosas. A diferencia de su dormitorio, es probable que sus programas sí sean examinados por otras personas.

Ejemplo 2.22:
Nombres de variables que pueden ser o no consistentes
a, b, c : si utiliza estos nombres para identificar las variables que almacenaran los datos para calcular el sueldo de un empleado, resultan no ser consistentes.
sueldoBase, descuento, horasExtras : estos nombres de variables sí identifican los datos que almacenarán y que tienen relación con el cálculo del sueldo para un empleado.

Ejemplo 2.23:
Buen uso de sangrías
El siguiente código fuente escrito en C++ muestra el buen uso de sangría para determinar las estructuras de control utilizadas.

```
 if (num >= 1) {
cout<<"  serie  de  ulam para"<<num<<endl;
while (num != 1) {
if (num% 2 == 0)
            num  = num /2;
        else
            num = num*3 +1;
            cout<< num<<endl;
}
}
 else
   cout<<"Ud. debe ingresar un número positivo mayor que cero"<<endl;
```

El uso de colores ha sido intencional para observar dónde se inicia y finaliza una estructura.

Otra recomendación es **utilizar una forma natural para las expresiones**. Esto significa escribir expresiones tal y como las diría en voz alta. Las expresiones condicionales que incluyen negaciones son siempre difíciles de entender.

Una primera recomendación es el **uso de sangrías para mostrar la estructura**. El uso de un estilo consiste en las sangrías, puesto que es la forma más sencilla de evidenciar la estructura de un programa. (ver ejemplos 2.23 al 2.26).

Ejemplo 2.24:
Uso natural de una expresión
El siguiente código muestra una manera de expresar una condición lógica que no es natural.

```
if (not num1 <1 && not num2 =0)
```

Una forma más clara de expresar lo mismo podría ser así.

```
if (num >= 1 && num2 =! 0)
```

Ahora el código se lee en forma natural.

Otra recomendación es el **empleo de paréntesis para resolver la ambigüedad**. Los paréntesis especifican agrupamiento y pueden ser utilizados para aclarar la intención, aun cuando no se requiera.

Ejemplo 2.25:
Uso de paréntesis para las expresiones

```
if ((num >= 1) && (num2 =0) || (num3 <6))
```

Otra recomendación es **dividir las expresiones complejas**. C, C++ y Java tienen muchos operadores y una rica sintaxis de expresiones, por lo cual resulta fácil dejarse llevar y escribir todo en una misma línea de instrucción.

Ejemplo 2.26:
Uso de expresiones complejas
El siguiente código fuente muestra una expresión compacta pero coloca demasiados operadores en una sola instrucción.

```
x + = ( xp=(2*k <(n-m) ? c[K+1] : d[k--]));
```

Es más fácil de captar cuando se fracciona en varias instrucciones.

```
if (2*k < n-m)
     xp = c[k+1];
else
    xp = d[k--];
x + = xp;
```

Ejemplo 2.27:

Uso de llaves y sangría para generar consistencia
El siguiente código fuente genera inconsistencia ya que no se comprende cuando inicia y finaliza un if.

```
if (mes == FEB) {
    if (anio % 4 ==0)
        if (dia >29)
            legal =FALSE;
    else
        if (dia >28)
            legal =FALSE;
 }
```

En el caso anterior la sangría es engañosa, porque el else en realidad está asociado con la línea

```
if (dia >29)
```

y el código está mal. Cuando un if sigue inmediatamente a otro, siempre deben emplearse llaves:

```
if (mes == FEB) {
    if (anio % 4 ==0){
        if (dia >29)
            legal =FALSE;
    } else {
        if (dia >28)
                legal =FALSE;
    }
 }
```

Por otra parte, si trabaja sobre un programa escrito por otra persona, conserve el estilo que éste tenga. Al hacer un cabio no use su propio estilo aunque lo prefiera. La consistencia del programa es más importante que la de usted, porque será más sencillo para quienes trabajan con él en el futuro.

c. Consistencia y convenciones

La consistencia da como resultado mejores programas. Si los formatos varían impredeciblemente, o un ciclo recorre un arreglo hacia adelante ahora y hacia atrás la siguiente vez, o si las cadenas se copian con strcpy aquí y con un ciclo for allá, las variaciones hacen más difícil ver lo que en realidad está sucediendo. Pero si la misma operación se realiza de igual manera cada vez que aparece, cualquier variación denotará una diferencia que valga la pena notar.

En este sentido se sugiere utilizar un **estilo consistente para las sangrías y las llaves**. Las sangrías muestran una estructura, pero, ¿cuál estilo es mejor?, ¿la llave que abre debe estar en la mima línea que el if o en la siguiente?. Los programadores han discutido siempre sobre el diseño de los programas, pero el estilo específico es mucho menos importante que su aplicación consistente. Elija un estilo, preferentemente el nuestro, úselo de manera consistente y no pierda el tiempo en discusiones.

Tal como los paréntesis, las llaves pueden resolver la ambigüedad y en ocasiones aclarar el código.

d. Comentarios

El propósito de un comentario es ayudar al lector de un programa. No se deben mencionar cosas que el código ya dice directamente, ni lo contradicen, tampoco se debe distraer al lector con elevados despliegues tipográficos o explicaciones. Los mejores comentarios ayudan al entendimiento de un programa indicando brevemente los detalles sobresalientes, u ofreciendo un punto de vista más amplio sobre los procesos.

Una buena recomendación es no utilizar comentarios para repetir lo que ya es obvio.

Ejemplo 2.28:
Uso de comentarios obvios
El código siguiente muestra el uso de comentarios que son completamente obvios.

```
/*
* caso de omisión
*/
default: break;

/* retorna 1*/
return 1;
```

En este caso, todos los comentarios deben eliminarse pues están demás.

Los comentarios deben añadirse para algo que no es evidente en el código, o reunir en un lugar información esparcida de todo el programa. Cuando algo sutil está sucediendo, un comentario lo puede aclarar.

Ejemplo 2.29:
Buen uso de comentarios

```
if ( c == '(')              /* paréntesis izquierdo*/
tipo = parteiz;
else if ( isdigit (c) )   /* número */
{    tipo = num;
     suma = totalAct   /*total actual de la operación según menú op 2 */
}
```

En este caso, los comentarios han sido utilizados para aclarar los resultados que no se esperaban.

Una buena práctica que también es recomendable es utilizar los comentarios para explicar una función y datos globales.

Ejemplo 2.30:
uso de comentarios en datos globales
El siguiente código fuente es un dato global ya que se escribe a continuación de las librerías.

```
struct producto  {  /* campos de un producto del supermercado */
    char  *nombre;
    int codigo;
    char *descrp;
}
```

Ejemplo 2.31:
Uso de comentarios para explicar una función
Aquí utilizamos comentarios para explicar para qué sirve la función factorial

```
/* función que entrega el factorial de n */
int function factorial (int n) {
Instrucciones para resolver el factorial de n
    return fact;
}
```

Finalmente, recomendaremos el uso de comentarios para aclarar de qué se trata un programa, el autor del mismo y la fecha de la última modificación. Este tipo de comentario lo incluiremos al principio de cada programa de tal manera que cualquier persona que habrá el archivo (código fuente) y lo lea, lo primero que encuentre es su descripción, permitiendo comprender de mejor manera el código escrito a continuación.

Ejemplo 2.32:
Uso de comentarios para describir un programa
El siguiente cometario explica la descripción de un programa, los datos del autor, la fecha de creación y la última fecha de modificación.

```
/*************************************************************
********
  Descripción: programa en lenguaje C++ que permite calcular el sueldo de
un empleado en base a las horas trabajadas y su grado asignado
  Autor(es): Ania Cravero, Mauricio Dieguez
  Fecha Creación: 15/07/10
  Fecha Ultima Modificación: 17/09/10
*************************************************************
********/
#include <iostream.h>
#include <math.h>
```

2.7.2 Depuración. Buenas pistas, errores fáciles de corregir

¡O no!. ¿Qué ocurrió?. Mi programa se cayó, o imprimió basura, o parece que se ejecutará para siempre. ¿Y ahora qué hago?.

Los nuevos programadores tienden a culpar al compilador, la biblioteca o cualquier otra cosa que no sea su código. Afortunadamente, la mayoría de los errores son simples y es posible encontrarlos con técnicas sencillas. Examine las salidas erróneas y trate de deducir cómo pudieron producirse. Revise las salidas de depuración antes de la caída, o haga una pausa en la línea de fallo para reflexionar ¿cómo pudo suceder eso? Razone el estado del programa en el momento que falló para determinar lo que pudo haber causado el problema

Cuando ya tengamos una explicación completa sabremos qué arreglar y, al mismo tiempo, podremos descubrir algunas otras situaciones inesperadas.

A continuación explicaremos algunas recomendaciones de errores comunes que podemos cometer al programar.

a. Busque patrones familiares

Pregúntese si éste es un patrón familiar. “He visto eso antes” suele ser el inicio del entendimiento del problema, o incluso la respuesta completa. Los errores comunes tienen síntomas característicos.

Ejemplo 2.33:

Errores comunes de programación

Por ejemplo, los programadores novatos de C muchas veces escriben

```
int n;
scanf (%d”, n);
```

en lugar de

```
int n;
scanf (%d”, &n);
```

y normalmente esto causa un intento de acceso a la memoria fuera de los límites al leer un dato. Quienes enseñan C reconocen el síntoma al instante.

Otro ejemplo de error común es cuando los programadores escriben las siguientes instrucciones de entrada/saluda en C++

```
cin<< a;
cout >> "este es " << a;
```

causando errores de sintaxis y por tanto el programa "no compilará" (en realidad detecta errores de sintaxis y no puede traducir a lenguaje de máquina). Estas líneas deben ser así.

```
cin>> a;
cout << "este es " << a;
```

Por último, otro ejemplo común donde encontramos errores es cuando no inicializamos alguna variable que haga de contador o sumador. El siguiente código no se ejecutará de manera correcta ya que la variable *suma* tomará cualquier valor extraño (que llamamos normalmente "basura") antes de acumular un nuevo valor.

```
for (i=0; i < 10; i++)
  suma = suma + i;
```

Cuando debiera ser

```
suma =0; //inicializamos el acumulador a cero
for (i=0; i < 10; i++)
  suma = suma + i;
```

b. Examine los cambios más recientes

¿Cuál fue el último cambio? Si sólo cambia una cosa a la vez a medida que el programa avanza, es muy probable que el error esté en el nuevo código o ha sido puesto en evidencia por este último. Una revisión cuidadosa a los cambios recientes ayuda a localizar el problema. Si el error aparece en la nueva versión y no en la anterior, el código nuevo es parte del problema. Esto significa que debe guardar al menos la versión previa del programa, que se supone es correcta, para poder comparar los comportamientos. También significa que debe registrar los cambios realizados y los errores corregidos, para no tener que redescubrir esa información mientras está tratando de corregir un error.

c. No cometa dos veces el mismo error

Después de corregir un error pregúntese si pudo haberlo cometido en alguna otra parte. Aún cuando el código pueda ser tan sencillo que lo pueda escribir dormido, no se duerma cuando lo escriba.

d. Corríjalo ahora, no después

Tener demasiada prisa también puede afectar en otras situaciones. No ignore una caída cuando ocurra; sígale la pista en ese momento, porque tal vez no ocurra sino hasta que sea demasiado tarde.

e. Lea antes de teclear

Una técnica de depuración efectiva, aunque poco apreciada, es leer el código con sumo cuidado y razonarlo por un momento sin hacer cambios. Por lo general, cuando ocurre un error la primera reacción es tomar el teclado y comenzar a modificar el programa para ver si el error desaparece. Pero es probable que ni siquiera sepamos lo que está mal y cambiar algo del programa equivocadamente puede traer mayores complicaciones y nuevos errores.

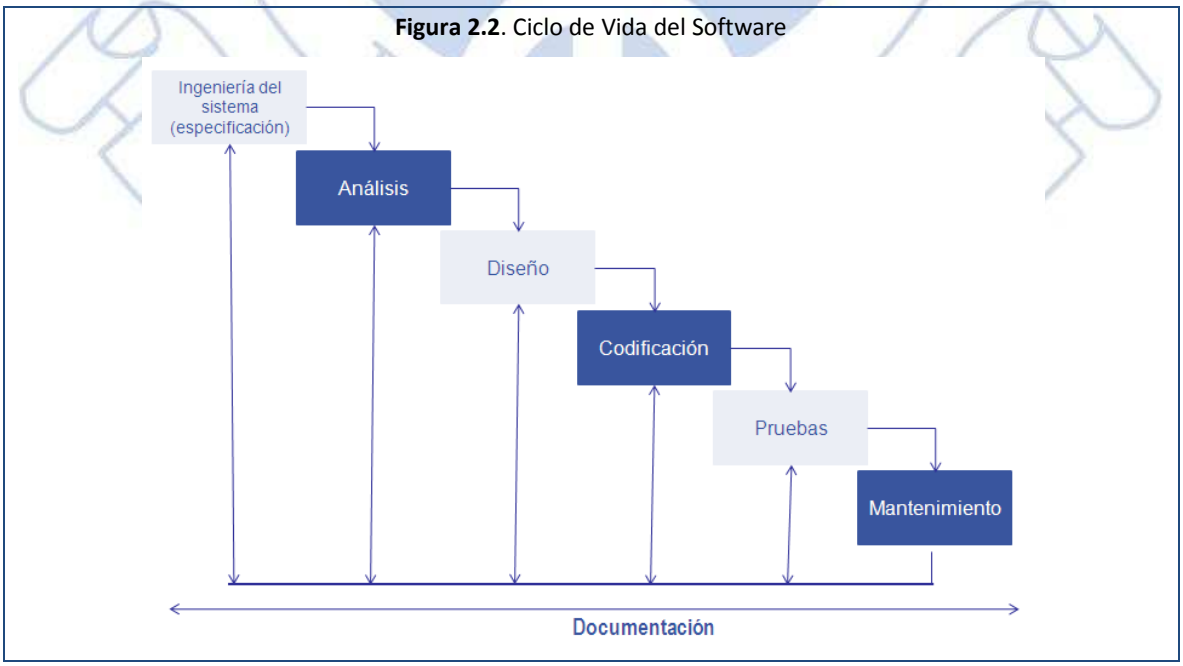
f. Explique su código a alguien más

Otra técnica efectiva es explicarle el código a alguien más. Muchas veces esto causará que nosotros mismos nos expliquemos el error. Algunas veces eso no toma más e unas frases, seguidas de un "No importa, ya sé lo que está mal. Perdón por la molestia".

2.8 El Ciclo de Vida del Software

Según José García-Bermejo (2008) la creación de programas es un proceso complejo. Esto hace que sea necesario fragmentar la tarea de programar el distintas etapas, que van desde la decisión de lo que debe hacer el programa hasta la corrección de errores y la adición de nuevas características. En su conjunto, el proceso de creación de programas recibe el nombre de Ciclo de Vida del Software, y consta de las etapas que muestra la figura 2.2.

Figura 2.2. Ciclo de Vida del Software



Todas las flechas del diagrama son bidireccionales, para denotar que el Ciclo de Vida del Software es un proceso iterativo. Esto significa, que un error que sea detectado en cualquiera de las etapas puede hacer necesario volver a cualquier otra etapa anterior. En este sentido, se debe considerar que cualquier error detectado en etapas posteriores son más graves que los detectados en las primeras etapas del proceso.

Los diferentes niveles de este modelo son los siguientes:

a. Ingeniería del sistema (Especificación de Requerimientos)

En esta etapa se lleva a cabo la especificación de las características y funciones que debe brindar el programa que se ha solicitado, así mismo las exigencias y normas que debe cubrir y la manera en que debe funcionar.

b. Análisis

Ya una vez que se cuenta con la llamada especificación de requerimientos, estos deben ser procesados, mediante un análisis, para intentar construir una estructura adecuada para describir todos los puntos de vista del problema. Así mismo se examina si existen soluciones alternativas, interfaces, se realiza una documentación formal de cada requerimiento y se analizan junto con el solicitante (futuros usuarios del software).

c. Diseño

Esta etapa sirve para especificar la forma en que se abordarán las distintas tareas. En esta etapa se debe decidir el procedimiento que se empleará para la realización de cada tarea. Esta actividad como la anterior, es previa a la selección del lenguaje de programación que será empleado. Observemos que pueden pasar semanas o meses en que no escribamos ninguna línea de código.

d. Codificación

Si el diseño se ha realizado de manera adecuada, la codificación debe tornarse transparente, esto no es más que un proceso en el que el diseño es traducido (dicho de manera informal) a un lenguaje que sea entendido por la máquina. Por lo tanto, lo primero que debemos hacer en esta etapa es seleccionar el leguaje de programación que dependerá de la plataforma de destino de la aplicación. El objetivo no es sólo obtener un programa correcto y robusto, sino que también un programa que se pueda desarrollar en equipo.

e. Pruebas

Una vez que se tiene el código de máquina de la aplicación, esta debe ser probada con datos reales, analizando el funcionamiento y lógica interna del programa, verificando que a cada entrada que se ha dado, esta genere los datos de salida esperados.

f. Mantenimiento

Una vez que un software es terminado, si se ha diseñado e implementado de manera adecuada, funcionará y permitirá que sea aceptable por un buen tiempo, pero a lo largo del tiempo y con el crecimiento de las exigencias del mundo real, se hacen necesarias modificaciones y en el peor de los casos se requieren modificaciones por errores encontrados. La etapa de mantenimiento (que puede suponer nada menos que el 60% de los costos de producción) se encarga de efectuar las oportunas reparaciones y mejoras, y de ir produciendo nuevas versiones del software. Esta es la etapa más larga del Ciclo de Vida ya que muchas de las ocasiones se aplica de por vida al sistema generado y conlleva a iteraciones dentro del modelo.

g. Documentación

Esta etapa abarca a todas las del proyecto, puesto que tiene como misión indicar el porqué de todas las decisiones tomadas y la forma de utiliza el producto (manual de usuario).

2.9 Comentarios Finales

En este segundo capítulo te hemos presentado una serie de conceptos básicos que te ayudarán a comprender la estructura de un programa.

En primer lugar hemos descrito de manera general un programa, los lenguajes de programación y su clasificación. Luego introducimos los conceptos básicos que deben ser comprendidos: *datos, variables, constantes, tipos de datos, variables y operadores*; para describir el proceso de creación de programas mediante una serie de herramientas de edición, traducción y ejecución. Con ello ahora debes ser capaz de comprender el proceso para crear un programa y luego ejecutarlo en el computador.

Finalmente, te describimos de manera breve el Ciclo de Vida del Software de manera de comprender de una manera simple las etapas de análisis, diseño, implementación y mantención de un programa

2.10 Referencias

José García-Bermejo (2008): "**Programación estructurada en C**". Pearson Prentice Hall. ISBN 978-84-8322-423-6

Roberto Carlos Guevara (2008): "**Sentencias básicas usadas en la programación e computadores**". Instituto Tecnológico Metropolitano. Fondo editorial ITM. ISBN 978-958-8351-57-5

Brian Kernighan y Rob Pike (2000): "**La práctica de la programación**". Pearson Prentice Hall. ISBN 968-444-418-4