Ingeniería en Desarrollo de Software
Primer Semestre

Programa de la asignatura:
Fundamentos de programación

Unidad 1. Introducción a la computadora y desarrollo de software

Clave:

TSU	Licenciatura
15141208	16141208

Universidad Abierta y a Distancia de México

Índice

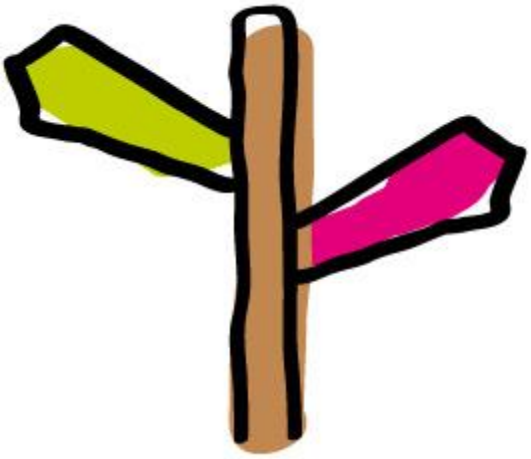
Unidad 1. Introducción a la computadora y desarrollo de software

Introducción

Bienvenido(a) al curso de *Fundamentos de programación*, en esta primera unidad estudiaremos qué son las computadoras y cómo pueden ayudarnos para resolver problemas.

Lo primero que debes tener claro es que las *computadoras* no poseen inteligencia alguna, ya que por sí solas no son capaces de resolver ningún problema, su importancia está en la capacidad de datos que pueden almacenar y manipular; de tal manera que, para lograr nuestro fin –resolver problemas mediante la computadora– es necesario desarrollar *programas* escritos en un *lenguaje de programación* para que puedan ser ejecutados por una computadora.

Propósitos de la unidad



- Identificarás los conceptos básicos relacionados con la computadora y los lenguajes de programación.
- Distinguirás los elementos de una computadora que intervienen en la ejecución de un programa a través del modelo de Von Neumann.
- Distinguirás los paradigmas de programación y los lenguajes asociados a éstos.
- Reconocerás las fases que se siguen para solucionar un problema mediante la computadora.

Competencia específica

Describir los elementos de la computadora y el ciclo de vida de software mediante el análisis un programa simple, con el fin de identificar los pasos que se realizan para construirlo y determinar qué elementos de la computadora intervienen en su ejecución.

Actividad previa de la unidad 1. Foro *Fundamentos de programación*

Con base en las instrucciones de tu docente en línea, **participa** en el Foro.

1. 1. ¿Qué es una computadora?

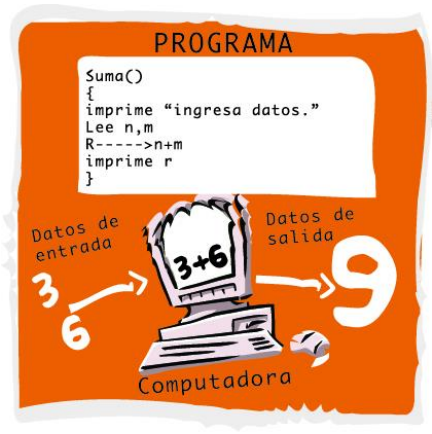
Antes de comenzar, te invitamos a observar el siguiente video en el que se hace una breve narración sobre la historia de la computadora, con el fin de brindar un panorama general de sus orígenes y de la manera en que ha evolucionado con el paso del tiempo hasta llegar al modelo actual de funcionamiento:

Da clic en el siguiente link para observar el video:
http://www.youtube.com/watch?v=88xNUNbPmEk&feature=player_embedded#!
Tomando en consideración la información presentada en el video, a continuación estudiaremos los siguientes temas que nos ayudarán a comprender mejor qué es y cómo funciona una computadora:

- El *Modelo de Von Neumann*
- Los pasos para realizar un programa
- Los principales paradigmas y lenguajes de programación utilizados actualmente.

Para fines de este curso entenderemos que una *computadora* es una máquina electrónica que recibe datos de entrada y los procesa de acuerdo al conjunto de instrucciones, llamado

programa, para obtener nuevos datos que son el resultado del proceso, tal como se ilustra en la siguiente figura:



Lo anterior nos lleva a clasificar los componentes de una computadora en dos clases: hardware y software.
Los recursos de *hardware* son todos aquellos elementos de la computadora que se pueden palpar, como por ejemplo: el monitor, el teclado, el disco duro, la memoria, entre otros.

Los recursos de *software* son aquellos elementos intangibles sin los cuales la computadora no funcionaría, esto es, el soporte lógico: programas y datos, entre los cuales se encuentran los sistemas operativos, editores de texto, compiladores, bases de datos, videojuegos, entre otros.

Como puedes darte cuenta, ambos elementos son necesarios para que una computadora funcione, ya que si los aislamos no pueden funcionar; por ejemplo, si tuviéramos una computadora con todos los componentes de hardware más avanzados que existen (procesador, tarjeta madre, monitor, teclado, etc.), pero sin Sistema Operativo instalado, de poco serviría pues al prenderla, únicamente veríamos una pantalla negra y, por consiguiente, no podríamos interactuar con ella. Para comprender mejor lo que acabamos de decir, avanza al siguiente tema en el que conocerás la forma en que se estructura y funciona una computadora.

1.2. Estructura y funcionamiento de una computadora

Te has planteado alguna vez preguntas como estas:

¿Cómo funciona y se estructura internamente la computadora?,
¿Cómo se obtienen los resultados? y ¿cómo se guardan los programas y datos en la memoria?

Cuando tenemos un primer acercamiento con equipos de cómputo, a pesar de manejarlos a diario, estas preguntas no son tan fáciles de contestar. Es por eso que en esta sección explicaremos y responderemos a estos cuestionamientos mediante el **Modelo de Von Neumann**, el cual constituye la base de la arquitectura de las computadoras actuales.

1.2.1. Modelo de Von Neuman

El Modelo de Von Neumann propone que tanto el programa como los datos sean almacenados en la memoria, de esta forma la computadora no tendría que reconstruirse, pues para programarla únicamente debe introducirse el programa por el dispositivo indicado, y posteriormente alimentar con los datos de entrada para que calcule la salida correspondiente.

Los elementos que componen esta arquitectura son: la *unidad central de procesamiento* integrada por la *unidad aritmética-lógica y la unidad de control*, la *memoria* y los *dispositivos de entrada/salida.* A continuación se describe brevemente la función de cada uno de los elementos que integran el Modelo de Von Neumann.

Unidad Central de Procesamiento*(CPU, Central Process Unit*), controla y coordina la ejecución de las instrucciones, para ello utiliza la Unidad Aritmético-Lógica encargada del procesamiento de los datos y la Unidad de Control para el procesamiento de las instrucciones.

- *Unidad Aritmético-Lógica(ALU, Arithmetic Logic Unit),* realiza todas las operaciones aritméticas (suma y resta) y lógicas (operaciones del Algebra de Boole). Además de los circuitos que le permiten realizar dichas operaciones, la ALU incluye un elemento auxiliar donde se almacenan temporalmente los datos que manipula conocido como *acumulador* o *registro temporal (TR, Temporal Register)*.
- *Unidad de Control(CU, Control Unit),* se encarga de leer las instrucciones almacenadas en memoria, decodificarlas y después enviar las señales a los componentes que están involucrados en su ejecución, para lo cual tiene dos elementos auxiliares el *Contador del Programa(PC, Program Counter)* y el *Registro de Instrucción(IR, Instruction Register)*. En el IR se guarda temporalmente la instrucción que debe ser ejecutada, mientras que en el PC se almacena la dirección de memoria que contiene la siguiente instrucción que se ejecutará.

Memoria principal*,* es la parte de la computadora donde se almacenan los datos y las instrucciones durante la ejecución de un programa. Físicamente está compuesta por circuitos integrados. Las computadoras actuales cuentan con un área de memoria de sólo

lectura – a la que se le conoce como memoria de tipo ROM (*Read Only Memory)* –y otra en la cual es posible escribir y leer datos – denominada de tipo RAM (*Random Access Memory*). La memoria RAM tiene el inconveniente de ser volátil pues al apagarse la computadora los datos almacenados se pierden.

Para resolver este inconveniente, se cuenta con otro tipo de memoria, denominada *memoria secundaria*, en ella se pueden almacenar una gran cantidad de información permanentemente, mientras el usuario no la borre. La desventaja de este tipo de dispositivos es que no son tan rápidos como la memoria RAM. Los discos duros, los discos ópticos (CD o DVD), la memoria flash (USB) y las cintas magnéticas, entre otras, son ejemplos de dispositivos de almacenamiento secundario.

Dispositivos de entrada y salida *(Input/Output),* son responsables de la comunicación con el usuario del sistema. Los *dispositivos de entrada* permiten introducir en la computadora datos e instrucciones, mismas que son transformadas en señales binarias de naturaleza eléctrica para almacenarlas en la memoria. Por otro lado, los dispositivos de salida permiten enviar los resultados a los usuarios de las computadoras, transformando las señales eléctricas binarias en información que éstos puedan comprender. El *teclado* está considerado como el *dispositivo de entrada estándar* pero existen otros del mismo tipo, por ejemplo: el ratón, el escáner, la lectora óptica, el micrófono o la tabla digital. A su vez el *monitor* es el *dispositivo de salida estándar*; otros ejemplos de dispositivos de salida son: impresora, bocinas, plotter, etc.

Es así que todas las unidades de la computadora se comunican a través del *sistema de buses* que son cables mediante los cuales se envían señales y dependiendo de la información que transmiten se clasifican en:

a) El bus de direcciones transmite la dirección de memoria de la que se quiere leer o en la que se quiere escribir.
b) El bus de control selecciona la operación a realizar en una celda de memoria (lectura o escritura).
c) El bus de datos transmite el contenido desde o hacia una celda de memoria seleccionada en el bus de direcciones según la operación elegida en el bus de control sea lectura o escritura.

Ahora ya sabemos cómo está estructurada internamente la computadora, qué elementos la integran y cuál es la función de cada uno de ellos, el siguiente paso es descubrir cómo colaboran para llevar a cabo la ejecución de un programa, en seguida lo explicamos:

Los datos de entrada que requiere un programa se introducen a la computadora, a través de los dispositivos de entrada; posteriormente se almacenan en la memoria RAM, para que la CPU pueda procesarlos, conforme a las instrucciones del programa, hasta obtener el resultado

deseado, mismo que envía al usuario por medio de los dispositivos de salida. Todas estas acciones son coordinadas por la unidad de control que envía las señales y datos a cada uno de los dispositivos de la computadora involucrados en la ejecución de las instrucciones del programa a través del sistema de buses. En la siguiente sección se describe con mayor detalle este proceso.

1.2.2. Ejecución de programas en la computadora

Para que entender mejor lo que sucede en el interior de la CPU al ejecutar cualquier programa, a continuación se describen de manera general los pasos que se realizan, una vez que el programa y los datos fueron almacenados en la memoria principal:

a) Primero, la unidad de control consulta en la memoria la instrucción indicada en el *contador del programa* y la almacena en el *registro de instrucciones*, actualizando el *contador del programa* con la dirección de memoria de la siguiente instrucción.

b) Después de que se almacenó la instrucción en el *registro del programa*, la *unidad de control* se encarga de decodificarla, detectando qué dispositivos están implicados en su ejecución, estos pueden ser: la ALU, cuando se tiene que hacer una operación; los dispositivos de entrada y/o salida, cuando se tiene que enviar o recibir un dato; o la memoria, si se quiere guardar o consultar un dato; posteriormente envía las señales de control a los mismos indicándoles la acción y, si es el caso, los datos y/o la dirección de memoria correspondiente.

c) Cuando los dispositivos realicen su tarea enviarán una señal a la unidad de control, para que esta repita el mismo procedimiento con la siguiente instrucción, así hasta ejecutar todo el programa.

Al período en el que se ejecuta una instrucción se le conoce como *ciclo de instrucción* o *ciclo fetch*.

Con el fin de ilustrar este *procedimiento***,** analizaremos la ejecución del siguiente programa escrito en un lenguaje de programación ficticio.

Ejemplo 1.1: El siguiente conjunto de instrucciones calcula el área de un rectángulo.

```
Imprimir “Ingresa la base:”
Leer   b
Imprimir “Ingresa la altura:”
Leer  h
area← b*h
Imprimir  área
```

Programa 1.1: Calcula área de un rectángulo

Antes de definir paso a paso la ejecución de este programa describiremos la función de cada una de las instrucciones que lo integran.

Instrucción	Descripción
Imprimir <Dato>	Imprime en el dispositivo de salida estándar los <Datos> indicados en la instrucción, que pueden ser un mensaje de texto o el valor de una variable.
Leer <X>	Lee por medio del teclado un dato, lo almacena en la variable <X> indicado y lo almacena en la memoria RAM.
<X> ← <Dato>	La flecha representa una **asignación**, esta acción actualiza la dirección de memoria asignada a <X> con el valor <Dato>.

Tabla 1.1: Lista de instrucciones en lenguaje de programación ficticio

Cabe señalar que en los lenguajes de programación, las direcciones de memoria se representan por medio de variables, para hacerlos más legibles. De tal manera que <X> representa una variable y <Dato> puede ser un mensaje o cualquier valor.

Ahora sí, de acuerdo con la información anterior, en la siguiente tabla se describen paso a paso las acciones que realiza la unidad de control junto con las otras unidades de la computadora involucradas en la ejecución de cada una de las instrucciones del programa.

Instrucción	Descripción de la instrucción
Imprimir “Ingresa base: ”	▪ La *unidad de control* envía señales al *monitor* para que imprima el mensaje “Ingresa base:”.
Leer b	▪ La *unidad de control* coordina las acciones necesarias para que, por medio del *teclado* ,el usuario introduzca un número y lo almacene en la *memoria principal*, en el espacio correspondiente a la variable *b*.
Imprimir “Ingresa altura:”	▪ La *unidad de control,* nuevamente, envía una señal al *monitor* para que imprima el mensaje “Ingresa altura:”.

Leer h	▪ La *unidad de control* coordina las acciones necesarias para que el usuario introduzca un número, por medio del *teclado,* y lo almacene en el espacio de *memoria* correspondiente a la variable *h.*
area← b *h	▪ La *unidad de control* envía la señal indicada a la *ALU* para que realice la multiplicación posteriormente envía la señal a la memoria junto con el resultado de la multiplicación, para que se almacene en el espacio de memoria *a.*
Imprimir area	▪ La *unidad de control* trae de la *memoria* el dato almacenado en el espacio asignado a la variable *area* y coordina las acciones para que el *monitor* imprima este valor.

Tabla 1.2: Ejecución paso a paso de un programa

1.2.3. Almacenamiento de programas y datos

La computadora sólo entiende señales binarias: ceros y unos, encendido y apagado, ya que todos los dispositivos de una computadora trabajan con dos únicos estados: hay corriente eléctrica y no hay corriente, respectivamente. Por tal motivo, los datos y programas almacenados en la memoria están codificados como cadenas de 1´s y 0´s para que la unidad de control pueda interpretarlos. A esta codificación se le llama *lenguaje de máquina.*

Cabe mencionar que la memoria está dividida en varias celdas, en cada una de ellas se puede almacenar únicamente 0s ó 1s, a estos valores se les denomina valores binarios o *BIT´s (BInary DigiT).*

Las celdas se agrupan para formar *registros* (también llamados *palabras*), a cada uno le corresponde una dirección de memoria, así cuando se desea escribir o leer de la memoria un dato o una instrucción se debe especificar la dirección dónde se encuentra.

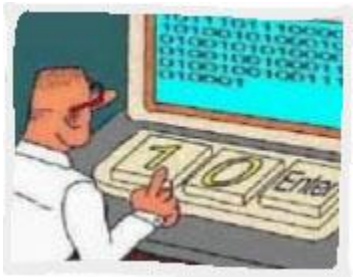
Como podrás imaginar, para un ser humano resultaría sumamente complicado escribir los programas en lenguaje de máquina, es por eso que los programas se escriben en lenguaje de programación entendibles para los seres humanos y después se traducen mediante un software especial –que pueden ser un compilador o un traductor– a cadenas de 0´s y 1´s. De tal manera que a cada instrucción le corresponde un código binario específico y para cada dato también existe una codificación única.

Por ejemplo, la palabra “Hola” se representa como “0100 1000 0110 1111 0110 1100 0110 0000”, ya que a cada letra le corresponde una codificación:

H	O	L	A
0100 1000	0110 1111	0110 1100	0110 0000

El número 80 se puede representar como “0101 0000” y la instrucción “MOV R2, R7” se codifica de la siguiente manera “0010 0000 1000 0100.”
Si quieres conocer más sobre el sistema binario, consulta la sección Material de apoyo.

Conforme fueron evolucionando las computadoras se inventaron diversas maneras de representar la información en código binario, hoy en día existen codificaciones estándar para los símbolos y los números, al igual que para las instrucciones; sin embargo, para nuestros objetivos es suficiente tener claro que cualquier dato o instrucción puede ser representado mediante cadenas de 0s y 1s.



Por otro lado, escribir programas en lenguaje binario es sumamente complicado para los seres humanos, por lo que en las últimas décadas se han desarrollado diversos lenguajes de programación que son más cercanos al lenguaje natural (humano), de los cuales hablaremos en la siguiente sección.

1.3. Lenguajes de programación

Los *lenguajes de programación* sirven para escribir programas de computadora orientados a resolver algún problema o necesidad. Cada lenguaje de programación se define a partir de un conjunto de símbolos básicos, llamado *alfabeto*; un conjunto de reglas, llamado *sintaxis*, que definen la forma de manipularlos o combinarlos para representar instrucciones; y las reglas que especifican los efectos de dichas instrucciones cuando son ejecutadas por la computadora, conocidas como *semántica*. De esta manera tenemos que:

$$Lenguajedeprogramación = alfabeto + sintaxis + semántica$$

Por otro lado, dependiendo de su legibilidad para el ser humano los lenguajes de programación se clasifican en *lenguajes de bajo nivel* y *lenguajes de alto nivel.* Los primeros se caracterizan porque sus instrucciones se parecen más a las acciones elementales que ejecuta una computadora, como son: sumar, restar, guardar en memoria, etc. En cambio, las instrucciones de los lenguajes de alto nivel son más parecidas a un lenguaje humano, por lo regular inglés. Por otro lado, los programas escritos en bajo nivel describen a detalle lo que sucede a nivel de hardware, mientras que los programas escritos en un lenguaje de alto nivel lo ocultan, teniendo como ventaja que son más fáciles de entender para las personas.

1.3.1. Evolución de los lenguajes de programación

Con las primeras computadoras surgió el primer lenguaje de programación que –como es de imaginarse– fue el *lenguaje de máquina,* el cual es considerado el lenguaje de primera generación. Las instrucciones en lenguaje de máquina dependían de las características de cada equipo, por lo que dada la dificultad de desarrollar programas en unos y ceros, los investigadores de la época desarrollaron el *lenguaje ensamblador*, cuyo conjunto de instrucciones consta de palabras nemotécnicas que corresponden a las operaciones básicas que una computadora puede ejecutar.

Para ilustrar esto revisemos la siguiente instrucción:[1]

Mueve el contenido del registro 8 al contenido del registro 10

En lenguaje de máquina esta se podría representar como:

0010 0000 1000 0100

Lo cual es ilegible para el ser humano, en cambio en lenguaje ensamblador esta instrucción se puede representar de la siguiente forma:

MOV R2, R7

Aunque sigue estando en clave, es más amigable que las cadenas de ceros y unos.

Otra característica del lenguaje ensamblador es que las instrucciones dependían de las características físicas (arquitectura) de la computadora.

Para traducir de lenguaje ensamblador a lenguaje de máquina, se desarrollaron programas llamados *ensambladores* (en inglés, *assemblers*). Este lenguaje fue considerado de segunda generación. Posteriormente, en la década de los 50´s aparecieron los primeros *lenguajes de alto nivel*, cuyas instrucciones son más parecidas al idioma inglés y, por lo tanto, más fácil de utilizar para los programadores, además de que son independientes de la arquitectura de las computadoras. Algunos ejemplos son: FORTRAN y COBOL (que son los primeros lenguajes que aparecieron y en sus inicios se utilizaron para aplicaciones científicas), C, Pascal, Ada, Lisp y Prolog (utilizados principalmente en inteligencia artificial), Java, C++, C#, entre otros.

Al igual que el lenguaje ensamblador, los programas escritos en un lenguaje de alto nivel deben ser codificados a lenguaje de máquina, así que junto con ellos se desarrollaron

[1] Este ejemplo es una adaptación de la versión original que aparece en (Joyanes & Zohanero, 2005, pág. 32) (Joyanes & Zohanero, 2005, pág. 32)

programas *traductores*, que de acuerdo con la forma en que trabajan se dividen en dos tipos: *compiladores* e *intérpretes*.

- Los *compiladores* traducen todo el programa escrito en un lenguaje de alto nivel, llamado *programa fuente*, generando un nuevo *programa objeto* que está escrito en lenguaje de máquina y a partir de este se genera un programa ejecutado, el cual puede ejecutarse cada vez que se desee sin tener que compilar el programa fuente de nueva cuenta. Además, como parte del proceso de traducción el compilador detecta los errores que hay en el código fuente, informándole al programador para que los corrija, pues un programa sólo se compila si no tiene errores.

- En cambio, un *intérprete* revisa una a una cada línea de código, la analiza y enseguida la ejecuta, sin revisar todo el código y sin generar un programa objeto, así que cada vez que se quiere ejecutar el programa se vuelve a traducir el programa fuente línea por línea.

Por lo anterior, los compiladores requieren una fase extra antes de poder generar un programa ejecutable, y aunque esto pareciera menos eficiente en cuanto a tiempo, los programas creados con compiladores se ejecutan mucho más rápido que un mismo programa ejecutado con un intérprete. Además, cuando un programa ya ha sido compilado puede ejecutarse nuevamente sin tener que compilarse, mientras que los programas que son interpretados, cada vez que se ejecutan se deben volver a traducir.

Conforme han ido evolucionando las computadoras también lo han hecho las estrategias para solucionar problemas, generando nuevos programas programación con diferentes filosofías, llamadas paradigmas de programación, de esto hablaremos a continuación.

1.3.2. Paradigmas de los lenguajes de programación

Un paradigma de programación representa un enfoque particular o filosofía para diseñar soluciones. Los paradigmas difieren unos de otros en los conceptos y la forma de abstraer los elementos involucrados en un problema, así como en los pasos que integran su solución del problema, en otras palabras, el cómputo.

Un lenguaje de programación siempre sigue un paradigma de programación, aunque también podemos encontrar lenguajes con la influencia de dos paradigmas, tal es el caso del lenguaje C++, que surgió bajo el paradigma procedimental y se transformó al paradigma orientado a objetos, de tal manera que puede soportar ambos paradigmas.
Los paradigmas más importantes son:

- *Paradigma imperativo o procedural.* Es el método de programación tradicional, donde los programas describen la forma de solucionar un problema a partir de una lista de instrucciones que se ejecuta de forma secuencial, a menos que se trate de estructuras de control condicionales o repetitivas, o bien, saltos de secuencia representados por la instrucción GOTO.[2] La programación imperativa se define a partir del cambio de estado de las variables que se produce por la ejecución de las instrucciones, por ejemplo, el programa 1.1, que calcula el área de un rectángulo, es un ejemplo de un programa imperativo, ya que describe paso a paso como solucionar el problema y el resultado corresponde al estado final de la variable *area*. Sin embargo, el lenguaje en el que está escrito no corresponde a ningún lenguaje de programación real, pero el lenguaje de máquina es un ejemplo de este paradigma. Otros lenguajes imperativos son: Fortran, Cobol, Pascal, Basic, Ada y C.

- *Paradigma declarativo.* En contraste con el paradigma imperativo, el objetivo de este paradigma no es describir cómo solucionar un problema, sino describir un problema mediante predicados lógicos o funciones matemáticas. Dentro de este paradigma se encuentran los lenguajes de programación funcionales y los lenguajes de programación lógicos. Los primeros representan el problema utilizando funciones matemáticas, por ejemplo, un programa que calcule el área de un rectángulo utilizando un lenguaje funcional se vería así:

$$areaRectangulo(b, h) = b * h$$

  De tal manera que para calcular el área de un rectángulo de base igual a 5 unidades y altura igual a 10 unidades, se ejecuta la función con los parámetros 5,10, es decir, areaRectángulo (5,10), la cual devuelve como resultado 50.

  Los lenguajes de programación más representativos del paradigma funcional son: Lisp, ML y Haskell.

  En el caso de los lenguajes lógicos la solución se representa a través de un conjunto de reglas, por ejemplo:

$$areaRectángulo(b, h, area) :- multiplicación(b, h, area)$$

  Esta regla dice que el valor de la variable *area* corresponde al área del rectángulo con base *b* y altura *h* sólo si *area* es el resultado de multiplicar *b* por *h***.** Estamos suponiendo que se ha definido el predicado *multiplicación(a, b, c)***.** En este caso para calcular el resultado se utiliza el principio de razonamiento lógico para responder a las preguntas planteadas, por ejemplo si se desea calcular el área del mismo rectángulo, la pregunta sería la siguiente:

$$? \; areaRectangulo(5,10, X)$$

[2] De esto hablaremos en unidades posteriores.

Y después de que se realicen los cálculos (en este caso llamadas inferencias) el resultado que arrojaría sería:

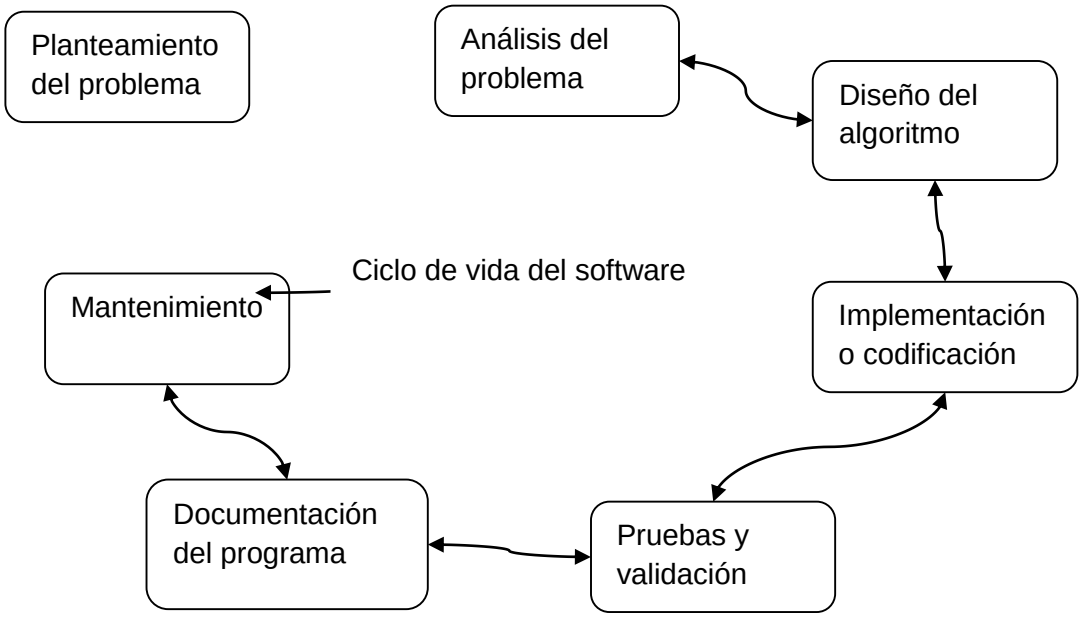
$$X = 50$$

El lenguaje más representativo del paradigma lógico es Prolog.

- *Paradigma orientado a objetos.* En este caso la solución de un problema se plantea en términos de objetos y relaciones entre ellos. Está basado en varias técnicas, incluyendo, herencia, polimorfismo, modularidad y encapsulamiento. En este caso se definen clases que son las plantillas para crear objetos, por ejemplo, un si se quiere un programa orientado a objetos que calcule el área de un rectángulo, se debe definir una clase rectángulo que contenga un método encargado de calcular el área. El lenguaje Java y C#, que actualmente son los más utilizados, son ejemplos de este paradigma.

1.4. Ciclo de vida del software

Independientemente del paradigma que se siga y del lenguaje que se utilice para programar, existe un conjunto de fases que deben seguirse para realizar un programa de computadora, al cual se le conoce como *ciclo de vida del software*, en la siguiente figura se lista cada una de ellas.



Planteamiento del problema

Es la primera fase del ciclo, consiste únicamente en elegir el problema que se quiere resolver para poder comenzar su análisis.

Análisis del problema

En esta fase se determina *¿qué hace el programa*? Por lo cual debe definirse de manera clara y concisa el problema en cuestión, se debe establecer el ámbito del problema, las características, limitaciones y modelos de lo que se desea resolver. Este paso debe conducir a una especificación completa del problema en donde se describa cuáles son los datos requeridos para resolverlo (*datos de entrada*) y cuál es el resultado deseado (*salida*).

El análisis de nuestro ejemplo es muy simple y se resume en la siguiente tabla:

¿Cuál es la salida deseada?	El área de un cuadrado, la cual identificaremos como
¿Qué método(s) se pueden utilizar para llegar a la solución?	El área de un rectángulo se puede calcular con la siguiente fórmula: $Área = Base * Altura$
¿Qué datos de entrada se requieren?	Por el planteamiento del problema y dado el método anterior, los únicos datos que se requieren son: la medida de la base que se representa por *b* y la medida de la altura indicada por *h*
¿Qué datos o información adicional es necesaria para solucionar el problema?	En este caso no se requiere más información.
¿Existe algún problema o condiciones que deban cumplirse?	Las únicas restricciones son que las medidas de la base y altura sean mayores a cero.

Tabla 1.3: Análisis del problema 1

Diseño de la solución

Es en esta fase se define *¿cómo el programa resuelve el problema*? Para ello, se describe paso a paso la solución del mismo, lo cual se conoce como *algoritmo*. Cuando el problema es grande se recomienda dividirlo en subproblemas más pequeños y resolver por separado cada uno de ellos. A esta metodología se le conoce como *diseño descendente (top-down)* o *modular.* Existen diferentes formas de representar un algoritmo algunas formales, como una fórmula matemática, o informales, como es el caso del lenguaje natural.

En la siguiente unidad estudiaremos a mayor profundidad los algoritmos y su representación, pero para seguir con el desarrollo de nuestro programa ejemplo, plantearemos la solución como una secuencia de pasos en español.

Algoritmo **Calcula el área de un rectángulo**	1. Obtener la medida de la base ($\boldsymbol{b}$) y la altura ($\boldsymbol{h}$) 2. Calcular: $\boldsymbol{área = b * h}$ **3.** Imprimir el resultado ($\boldsymbol{área}$)

El programa 1.1 es otra forma de representar la solución de este problema, se conoce como pseudocódigo.

Implementación (codificación)

El algoritmo no puede ser ejecutado por una computadora por ello debe traducirse a un *lenguaje de programación* (como por ejemplo C) para obtener un *programa fuente* que se traduzca a lenguaje de máquina para que sea ejecutado por la computadora.

En el siguiente cuadro se muestra la codificación en lenguaje C del algoritmo, por ahora no es necesario que lo comprendas puesto que esto lo podrás hacer conforme vayas aprendiendo a programar, por lo pronto solamente se muestra con fines ilustrativos.

```
#include<stdio.h>
#include<stdlib.h>
main()
{
    int b,h, area;
printf("Ingresa la base y altura:");
scanf("%d %d", &b,&h);
    area = b * h;
    printf("Area = %d", area);
}
```

Programa 1.2: Programa en C que calcula área de un rectángulo

Validación y pruebas

Esta fase debe hacerse una vez que se ha diseñado el algoritmo y después de que se codifica, sirve para verificar que son correctos. Existen diferentes formas de probar que la solución es correcta, algunas de ellas formales y otras informales: las primera se utilizan para garantizar que el programa o algoritmo siempre calcula el resultado deseado para cualquier conjunto de datos de entrada; en cambio, en las segundas sólo se prueba que funciona correctamente para

algunos datos de entrada, tratando de encontrar posibles errores, en este caso no se puede garantizar el programa o algoritmo calcule la salida correcta para cualquier conjunto de datos. En cualquiera de los dos casos, si se encuentra alguna falla se debe corregir y volver a realizar pruebas. En este curso utilizaremos las pruebas de escritorio, las cuales se explicarán en la unidad 2.

El ejemplo es muy sencillo y si ejecutamos manualmente el programa o algoritmo mostrado en la fase anterior, con un caso específico de rectángulo veremos que ambos son correctos. En la siguiente figura se ilustra la ejecución del programa:

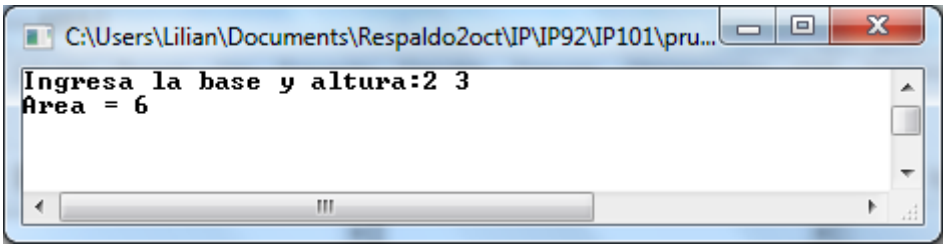


Figura 1.1: Ejecución del programa 1.2

Documentación

Cualquier proyecto de software por la complejidad que tiene requiere tanto las ideas principales como el desarrollo de principio a fin sea documentado, con el fin de que cualquiera puedan entender la lógica del programa y de ser necesario pueda modificarlos sin tantas complicaciones. Es común que si se desea modificar un programa y no se tiene información acerca de cómo fue construido sea más fácil volverlo a hacer que intentar entenderlo. Uno de los mejores ejemplos de la importancia de la documentación es el software libre, en el cual colaboran diversos desarrolladores para su elaboración, los cuales se encuentran en diferentes puntos geográficos de globo terráqueo, así que la forma de entender que está haciendo cada uno y bajo que método es la documentación. Además de que se debe tomar en cuenta que se llama software libre porque está disponible el código fuente para que cualquier persona pueda modificarlo a su conveniencia.

Como parte de la documentación también deben incluirse manuales de usuario y las normas de mantenimiento para que se haga un buen uso del software.

Mantenimiento

Esta fase tiene sentido una vez que fue terminada una primera versión del programa y ya está siendo utilizado. Ya que en ésta se actualiza y modifica para corregir errores no detectados o para cambiar y/o agregar una nueva función. Por ejemplo, se puede extender el programa 1.1, que calcula el área de un rectángulo para que también calcule su perímetro.

Ejemplo 1.2: El siguiente conjunto de instrucciones calcula el área y perímetro de un rectángulo.

```
#include<stdio.h>
#include<stdlib.h>
main()
{
      int b,h, area, perimetro;
printf("Ingresa la base y altura:");
      scanf("%d %d", &b,&h);
perimetro = 2*b + 2*h;
area = b * h;
printf("Perimetro = %d", perimetro);
      printf("Area = %d", area);
}
```

Programa 1.3: Calcula el área y perímetro de un rectángulo

En el programa se resaltan las instrucciones que se añadieron al programa para calcular el perímetro.

Cierre de la Unidad

Aquí concluimos la primera unidad de nuestro curso en la que, de manera general, podemos decir que aprendimos las partes que integran una computadora y la manera en que cada una de ellas interactúa para obtener como resultado la interacción con el usuario.

También aprendimos que existen varias formas de comunicarnos o hacernos entender por estas máquinas a través de lo que denominamos *Lenguajes de programación* y que existen varios paradigmas que determinan las características de esos lenguajes.

Por último, conocimos las fases que se siguen cuando se realiza un programa de computadora, independientemente del lenguaje que se utilice.

Si todo esto que acabamos de decir te parece ya familiar, es decir, si sabes a que nos estamos refiriendo al mencionar lo anterior, entonces ya estás lista(o) para continuar con la unidad dos.

¡Adelante!

Fuentes de consulta

- Guerrero, F. (s.f.). *mailxmail.com.* Recuperado el 15 de agosto de 2010, de http://www.mailxmail.com/curso-introduccion-lenguaje-c

- Joyanes, L., & Zohanero, I. (2005). P*rogramación en C. Metodología, algoritmos y estructuras de datos*. España: Mc Graw Hill.

- Reyes, A., & Cruz, D. (2009). *Notas de clase: Introducción a la programación*. México, D.F.: UACM

- Viso, E., & Pelaez, C. (2007). I*ntroducción a las ciencias de la computación con Java*. México, D.F.: La prensas de ciencias, Facultad de Ciencias, UNAM.