



De 0 a C Developer

wozgeass

---

# Table of Contents

Léeme	1.1
Licencia	1.2
Apéndice	1.3
Historial de Cambios.	1.4
Información Personal	1.5
Iniciando con la Programación.	1.6
Introducción	1.6.1
Metodología de la programación	1.6.2
Editores	1.6.3
Relación computadora y los lenguajes	1.6.4
Introducción al lenguaje C	1.7
Buenas Practicas	1.7.1
Historia	1.7.2
Introducción	1.7.3

# De-0-a-C-Developer

Un libro para aprender a programar iniciando con el lenguaje **C** usando sistemas **Linux** y ser capaces de contribuir o crear proyectos **Open Source** que ocupen este lenguaje.

Este lenguaje es usado en proyectos como:

Proyectos
Gnome
Linux Kernel
Gnu
Hurd Kernel
Xfce
Lxde
Vim
The Gimp
Inkscape
Audacity
Vlc
Postfix
Entre muchos mas ...

Y si aun permaneces escéptico pueden verificar directamente buscando el código fuente de cada proyecto.

**Bienvenido a lo bonito.**

A terminal window with a dark background and a light gray border. The text ">gcc" is displayed in white, with a small white glow effect behind the greater-than sign.

```
>gcc
```

# Distribución

## Distribución del Texto

Atribución-Compartir Igual CC BY-SA

Esta licencia permite a otros re-mezclar, retocar, y crear a partir de tu obra, incluso con fines comerciales, siempre y cuando te den crédito y licencien sus nuevas creaciones bajo las mismas condiciones. Esta licencia suele ser comparada con las licencias "**copyleft**" de software libre y de código abierto. Todas las nuevas obras basadas en la tuya portarán la misma licencia, así que cualesquiera obras derivadas permitirán también uso comercial. Esa es la licencia que usa Wikipedia, y se recomienda para materiales que se beneficiarían de incorporar contenido de Wikipedia y proyectos con licencias similares.



## Distribución de los código de ejemplo

Para los códigos que se vayan haciendo como ejemplos tendrán una licencia "**GPL v2**"

This program is free software; you can redistribute it and/or modify it under the terms of the GNU General Public License as published by the Free Software Foundation; either version 2 of the License, or (at your option) any later version.

This program is distributed in the hope that it will be useful, but WITHOUT ANY WARRANTY; without even the implied warranty of MERCHANTABILITY or FITNESS FOR A PARTICULAR PURPOSE. See the GNU General Public License for more details.

You should have received a copy of the GNU General Public License along with this program; if not, write to the Free Software Foundation, Inc., 51 Franklin Street, Fifth Floor, Boston, MA 02110-1301, USA."

# Motivaciones

Me animé a realizar este libro con el fin de mostrar el fascinante mundo de la programación usando el lenguaje **C**, para que las personas que no lo conocen, lo entiendan de una manera simple y con muchos ejemplos.

Este es la primera parte de 3 más que le precederán cuya finalidad será que podamos sumergirnos en cada momento a más detalle sobre el lenguaje y como poder programar cualquier cosa.

Debo admitir que este libro sera un poco largo y en algunos momento bastante tedioso, tratare de explicar muchas partes con el mayor detalle posible ya que la otra finalidad de este libro es aprenderlo lo mejor que se pueda.

Si llegaras a encontrar algo que no esta bien agradecería que me lo notificaras lo antes posible, con la finalidad de mejorar la calidad del contenido de este libro.

## ¿A quien esta dirigido?

Para personas que sepan muy poco o nada de programación y que les interese iniciar a resolver problemas y plantear su solución en lenguaje **C**.

Si alguna vez viste algún otro libro podrás haberte dado cuenta que están muy atrasados o en ingles, mi enfoque es presentarte la información de la manera mas didácticamente posible, explicándote como resolver tus problemas usando librerías y tecnologías **Open Source**.

Considero que esta característica es fundamental, sobre todo desde el punto de vista académico, por que trata de enseñar, de hacer entender, de hacer ver, al lector, como resolver un problema, y luego como programar esa solución en **C**.

Y también para los que ya tienen ciertas nociones y quieran mejorar, repasar un poco o quizás aprender algo nuevo les va a ser muy útil este libro.

Considero que esta es una verdad que aplica también en este campo:



"Cualquiera puede programar".

Es importante destacar que el nivel de complejidad de los temas aumenta en forma gradual, y cada uno tratare de hacerlo con la mayor amplitud y claridad posible.

Todos queremos sentirnos relevantes, y pensar que estamos haciendo una diferencia en este mundo. Tener un trabajo donde uno se siente productivo, y saber que el trabajo que uno hace “importa” es un gran desafío. *Linus Torvalds*

## Recomendaciones

Utilizaremos **Linux** como sistema operativo base para correr todos los código que se hagan, ademas agregare un apartado detallando el compilador **GCC 6.1** que sera el que usaremos para compilar todos los ejemplos.

# Historial de Cambios

# de Versión	Fecha	Editor	Comentarios
0.1	Julio 2015	Adrián	Comienzo
..	..	..	..



## Información de Contacto



Mi nombre es **Adrián** me he enfocado en el estudio los aspectos técnicos de varias tecnologías, soy amante de las ideologías *Open Source* sin embargo también disfruto los aspectos técnicos de muchas cosas, incluso componentes internos del sistema operativo *Windows*, *MacOSX*, *FreeBSD* y **Android**.

Lo que mas llama mi atencion es la programación a bajo nivel y el estudio exhaustivo de **Linux** desde sus partes mas internas, por lo que he realizado y asistido a múltiples conferencias sobre estos temas y también realizo recuentos detallados de los cambios que ha tenido el **kernel linux** en cada release nuevo liberado.

Espero encuentres este libro muy ilustrativo y disfrutes leyendolo casi tanto como yo haciéndolo, por lo que iniciemos formalmente con lo emocionante.

Si deseas contactarme puedes enviarme un correo a [adrian@aztli.org](mailto:adrian@aztli.org)

**XD.**



# Iniciando con la Programación

En esta parte aprenderás varios conceptos básicos, entre ellos:

- ¿Qué es un lenguaje de programación?
- ¿Qué es programar?
- ¿Qué es un algoritmo?
- ¿Qué es un paradigma?
- ¿Qué es un diagrama de flujo?
- ¿Qué es el pseudocódigo?
- Las buenas practicas en la programación
- Editor de Texto a utilizar.

# Introducción

Cuando ocupas una computadora siempre te cuestionas que es lo que la hace funcionar, como es que puedes hacer tu trabajo. Para todo esto es importante tener nociones de programación, ya que siempre podremos encontrarnos con distintos problemas que como tales son muy difíciles de solucionar.

En muchos de estos casos el aprender un lenguaje de programación nos ayuda a ocupar el poder de procesamiento de la computadora para la solución de estos problemas aparentemente imposibles de solucionar, por otra parte te va a permitir experimentar el gozo de haber creado algo por mismo o inclusive el mismo gusto pero de haber contribuido a algún proyecto **Open Source**.

Cabe destacar que la complejidad de la programación dependerá del problema que se trate de solucionar, comprender que toda esta idea de poder dar solución a problemas con mayor rapidez y eficacia se debe en gran parte a los avances y progresos en electrónica, matemáticas, física, etc.

En mi opinión la programación no es algo tan difícil, para muchos de nosotros que no tuvimos ciertas nociones, empezar con algo así no fue fácil, pero no por su complejidad, si no mas bien por que tuvimos un mal comienzo y quizás quisimos correr en lugar de aprender buenos fundamentos..

Un poco mas tarde me di cuenta que lo que distingue a muchos programadores de otros es su creatividad, para poder iniciar el desarrollo de algo tan simple o tan complejo como su mente visualice.

Por lo que la programación es una disciplina que requiere simultáneamente del uso de cierto grado de creatividad, un conjunto de conocimientos técnicos asociados y la capacidad de operar constantemente con abstracciones (tanto simbólicas como enteramente mentales).

La creatividad necesaria para programar no se diferencia demasiado de aquella utilizada para producir textos. Sin embargo, lo que hace a la programación algo especial es que requiere emplear un conjunto de conocimientos técnicos asociados a la manipulación de las computadoras. Esto agrega un grado notable de rigurosidad a esta actividad, ya que no podemos programar sin tener en cuenta este aspecto. Por otra parte, al poseer una naturaleza ligada a la resolución de diferentes problemas del mundo real, se requiere de una capacidad de abstracción que permita operar sin que los conocimientos técnicos limiten al programador a resolver adecuadamente dichos problemas.

Por ejemplo actividades que requieren:

Un uso intensivo de la creatividad son relacionadas comúnmente con el Arte.

Conocimientos técnicos profundos son los relacionados con la medicina, electrónica y química.

Operar continuamente en abstracto son las relacionadas con filosofía, lógica y matemáticas.

A lo largo de la vida seguiremos enfrentando continuamente con todo tipo de problemas. Para ellos nos valdremos de distintas herramientas, que combinadas de maneras innovadoras ampliaremos el espectro de soluciones y vuelven factible el desarrollo de cualquier programa.

Algo muy cierto es que a los que les guste la programación se podrán dedicar a construir programas, mantenerlos o contribuir a diversos Proyectos y siempre habrá mercado para esas personas.

## Conceptos de la programación

Lo que nos lleva a definir formalmente que es un programa:

Un **Programa** es una descripción ejecutable de soluciones a problemas computacionales, es decir, un texto descriptivo que al ser procesado por una computadora da solución a un problema propuesto. De esta manera, la parte descriptiva de los programas es el texto que el programador le provee a la computadora.

Y como dijimos antes las descripciones dadas por los programas pueden estar escritas con diferentes símbolos y para diferentes propósitos.

Cuando el código consiste mayormente de palabras y nociones que son mas sencillas para que manejen las personas, con el objetivo de que puedan entender y construir los programas, hablamos de un lenguaje de **Alto Nivel**, y al código resultante lo llamaremos **código de fuente**.

Cuando el código consiste mayormente de números y símbolos de difícil comprensión para las personas, pero de rápida ejecución por una maquina, hablamos de un lenguaje de **bajo nivel**, y al código resultante lo llamamos **código objeto** o **código ejecutable**. Por lo que hablaríamos de un **alto nivel de abstracción** cuando nos referíamos a la abstracciones

mas cercanas a las ideas del problema a ser solucionado, a la mente de los programadores; y de **bajo nivel de abstracción** cuando nos refiramos a abstracciones mas cercanas a las ideas relacionadas a la forma de funcionamiento de las computadoras.

Es importante observar que tanto el código fuente como el código ejecutable están conformados por símbolos, y en este sentido es correcto llamar a ambos programas. Esto suele crear cierta confusión, pues entonces la palabra programa se utiliza para dos propósitos diferentes:

1. El código que escribe el programador.
2. El código que ejecuta la computadora.

El primer punto sera el punto de estudio en este libro y el segundo es el resultado de varios procesos de traducción y compilación sobre el código fuente, que veremos un poco mas adelante en un apartado especial.

Si bien cuando escribimos el código fuente de un programa utilizamos símbolos como los del lenguaje natural, este texto debe ser capaz de poder ejecutarse por medio de la traducción que realiza nuestro compilador. Siendo esta característica la que hace que los programas se diferencien de otros textos, ya que no cualquier texto es ejecutable por una computadora. Lo hace a un texto ejecutable es mas bien su *sintaxis dura*, que no es mas que un conjunto de reglas estrictas de un determinado lenguaje de programación. con las que se escribe el código fuente.

Los programadores tienen que balancear dos mundos bien diferente: un mundo de estructura y un mundo de imaginación. Creando conceptos abstractos usando lenguajes de programación muy estructurados. [Programmer-Creativity-Boost](#)

## Lenguajes de Programación

Cuando programamos, no podemos utilizar el lenguaje natural con que nos comunicamos cotidianamente. Por el contrario, se emplea un lenguaje que un interprete o un compilador sea capaz de traducir a lenguaje maquina.

Un lenguaje de programación es una serie de reglas que establecen que descripciones serán aceptadas y ejecutadas y cuales no tienen sentido para el mecanismo de ejecución provisto por la computadora. Además, estas reglas están diseñadas de manera composicional, para que sea sencillo construir programas de mayor envergadura.

En actualidad existe un gran numero de lenguajes de programación, de características sumamente diferentes y que emplean diferentes enfoques con los que podemos programar. Cada lenguaje depende de un conjunto de ideas que guían, como su propósito final, la forma en que codificamos la descripción que otorgaremos a la computadora. Ya que esta forma viene dada por las reglas que definen como se combinan los elementos que el lenguaje de programación provee al programador.

A su vez, algunos lenguajes de programación están pensados para volcar mejor las ideas abstractas que el programador intenta emplear. Esto se conoce como lenguaje de **alto nivel**, ya que intentan, con cierto grado de eficacia, soslayar aquellas tareas que la computadora requiere realizar para que el programa cumpla con sus objetos. Pero también existen lenguajes de **bajo nivel**, que no hacen definir en mayor o menor medida cada paso que el programa seguirá, y por ende, están ligados a la naturaleza operacional de la computadora. Esta denominación (**Bajo y Alto**) surge de la idea de imaginar que la computadora es la base fundacional sobre la que los programas se ejecutan, que de las ideas abstractas se van construyendo sobre ellas. Justamente denominados a un lenguaje de mas **alto nivel** de abstracción cuanto mas lejos de la base se encuentra, o sea, cuanto mas abstracto, menos relacionados con el proceso real de ejecución.

Un lenguaje de **Alto nivel** expresa mejor las ideas en las que debe pensar el programador, y en alguna forma están mas alejado de la maquina especifica que ejecutara cada programa escrito en dicho lenguaje.

Un lenguaje de **Bajo Nivel** expresa mejor las ideas propias de los mecanismos de ejecución, por lo que es mas dependiente de la maquina especifica que ejecutara cada programa escrito en dicho lenguaje.

En la gran mayoría de los cursos de programación inicial suelen utilizarse una forma de lenguaje denominado comúnmente pseudocódigo, que se confunde con la de lenguaje de programación. Pero que no veremos por motivos de que seria muchísimo mas extenso este libro.

## Paradigmas de programación

Durante la década de 1960, la proliferación de lenguajes de programación siguió creciendo, y de a poco fueron diferenciándose grupos o familias de lenguajes, en torno a la predominancia de ciertas características. El foco de desarrollo de software se fue desplazando hacia la educación, para poder formar a los futuros programadores. La administración a gran escala siguió teniendo fuerte presencia, pero las aplicaciones orientadas a la defensa fueron disminuyendo.

Estos grupos o familias de lenguajes dieron origen a lo que ahora denominamos paradigma de programación, que no es otra cosa que un conjunto de ideas y conceptos al respecto del estilo con el que se expresan las soluciones a problemas a través de un lenguaje de programación. Cada paradigma privilegia ciertas ideas sobre otras, y ciertas formas de combinación por sobre otras, dando lugar a estilos muy diferentes en la forma de programar.

Un **paradigma de programación** es un conjunto de ideas y conceptos vinculados a la forma en que se relacionan las nociones necesarias para solucionar problemas con el uso de un lenguaje de programación.

Para 1970 ya se podían identificar cuatro grandes paradigmas, que están vigentes hoy incluso hoy en día y que son claramente reconocidos. Mucho después se intentó identificar paradigmas adicionales a estos cuatro, pero no hubo cierto consenso sobre si alguno de ellos llegaría o no a poder ser considerado un paradigma y merecer ese nombre.

Los cuatro paradigmas de programación que surgieron a fines de los **60s** y principio de los **70s**, y que resultaron de fundamental influencia en la forma de hacer la programación, son:

1. El paradigma *Imperativo*.
2. EL paradigma *Orientado a Objetos*.
3. El paradigma *Funcional*.
4. El paradigma *Lógico*.

Los dos primeros están mas orientados a la forma de manejar estados y podrían ser denominados **procedurales**, mientras que los 2 últimos están mas orientados a expresar conceptos o nociones independientes del estado y podrían ser denominados **declarativos**. El paradigma que se desarrollo con mayor ímpetu al principio fue el imperativo, debido a su cercanía con los lenguajes de bajo nivel. Los otros tardaron mas tiempo en adoptar un estado de madurez, y no fue hasta mediados de la década de **80s** que tanto el paradigma funcional como el lógico y el orientado a objetos empezó a ser foco de atención masiva.

Dentro del paradigma imperativo se clasifican lenguajes mas vinculados con la secuencia de instrucción y mas cercano al ensamblador. Algunos nombres notables que surgieron en esa época dentro del paradigma imperativo, y aun conocidos hoy en día son:

- **Basic**, desarrollado en 1965 por John Kemey y Thomas Kurtz con la intención de que se convirtieran en un lenguaje de enseñanza.
- **Pascal**, desarrollado en 1970 con fines didácticos, por Niklaus Wirth a partir de Algo.
- **C**, desarrollado por Dennis Ritchie y Ken Thompson en los laboratorios Bell entre 1969 y 1973, con el propósito de proveer una traducción eficiente a ensamblador y permitir la administración eficaz de los recursos de computo de las maquinas con arquitectura Von Neumann a través de abstracciones cercanas al bajo nivel, que brinda una forma cómoda e independiente a la computadora de administrar sus recursos.



Y la fama y el motivo en el que nos centraremos en ese lenguaje se debe a que por la característica anterior en su diseño, fue utilizado en la programación del sistema operativo **UNIX** y fue ampliamente portado a numerosos sistemas. Es uno de los lenguajes mas difundidos y conocidos de todos los tiempos, y su estudio implica un conocimiento profundo de la forma en que se ejecutan un gran numero de aplicaciones en **Linux**, **Unix** y otros sistemas operativos.

Dentro del paradigma funcional se clasifican lenguajes orientados a la descripción de datos, de su forma, las relaciones entre ellos, y sus transformaciones. Si bien inicialmente no fueron tan populares, la investigación llevo a este paradigma a la madurez y desde el mismo se realizaron grandes aportes a todos los lenguajes modernos. Algunos lenguajes que surgieron en esa época dentro de este paradigma son:

- **ML**, desarrollado por Robin Milner y otros a principios de los 70s en la universidad de Edimburgo en Reino Unido con el propósito de servir para desarrollar tácticas de prueba en herramientas de demostración automática de teoremas, utilizando un sistema de tipos estáticos que es una de sus grandes innovaciones.
- **Miranda**, desarrollado por David Turner en 1985 como sucesor de sus primeros lenguajes de programación SASL y KRC, incorporando conceptos aprendidos del lenguaje ML y Scheme, derivado como dialecto de LISP por Guy L. Steele and Gerald J. Sussman en el laboratorio de Inteligencia artificial del MIT, siguiendo principios de minimalidad en la cantidad de conceptos distintos a proveer, pero conservando un gran poder expresivo.
- **Haskell**, en Honor de Haskell B. Curry, se publico por primera vez en 1990, y es actualmente el lenguaje de alto nivel con mayor pureza conceptual, expresando el estado del arte en el desarrollo de lenguaje de programación funcional. Su impacto en la comprensión de conceptos de alto nivel no puede ser ignorada por un programador actual.

Dentro del paradigma orientado a objetos se encuentran lenguajes que agrupan el código alrededor de la metáfora de objeto, y que intenta representar mediante datos encapsulados las entidades del mundo real. Al ya mencionado lenguaje SIMULA, pionero de los lenguajes orientados a objetos, y algunos otros como:

- **SmallTalk**, creado en el Learning Research Group de Xerox por Alan Kay y Otros, también en los 70s, Pensado con fines educacionales basándose en la teoría constructivista del aprendizaje. Siendo la base del desarrollo posterior en tecnología de objetos, que hoy es uno de los pilares de la construcción moderna de software.

Seguramente te preguntaras sobre los lenguajes de programación multi propósito como java, python, c++, puestos son lenguajes de programación que son considerados con mas de un paradigma por lo que a pesar de tener el soporte de la orientación a objetos, no los incluiré.

Finalmente, dentro del paradigma lógico se encuentra distintos lenguajes orientados a la descripción de las relaciones lógicas entre aserciones, que habilitaban las posibilidad de realizar inferencias y ciertas formas de razonamiento automático, lo cual fue la inspiración para desarrollar el área conocida como inteligencia artificial. El lenguaje mas conocido de este paradigma, que también surgió a finales de la década de 1970 en Marseille, Francia, en el grupo de Alain Colmerauer, es PROLOG, un lenguaje basado en la afirmación de hechos y reglas de inferencia, que se utilizan mediante consultas en la forma de afirmaciones que deben validadas a partir de los hechos.

Seguramente te veras que ademas de Prolog existe otro lenguaje llamado lisp, ese lenguaje a pesar de ser lógico también puede ser ocupado con otros tipos de paradigma sin embargo se le conoce mas por ser ocupado con paradigma lógico.

Cada uno de los lenguajes mencionados dejo un sin numero de descendientes y prácticamente todos los lenguajes modernos se vinculan, de una forma u otra, con algunos de estos.

Otro gran avance de esta época fue el desarrollo de lo que dio en llamarse programación estructurada, que sucedió dentro del paradigma imperativo, y consistió fundamentalmente en aumentar la abstracción de los lenguajes, eliminando primitivas de control des-estructurado, o sea, que permitan moverse libremente por el código, sin tener en cuenta su estructura lógica.

## Algoritmo

Las personas efectuamos cotidianamente series de pasos, procedimientos o acciones que nos permiten alcanzar algún resultado o resolver algún problema. Estas series de pasos, procedimientos o acciones, comenzamos a aplicarlas desde que empieza el día, cuando, por ejemplo, decidimos bañarnos. Posteriormente, cuando tenemos que ingerir alimentos también seguimos una serie de pasos que nos permiten alcanzar un resultado específico. Y la historia se repite un innumerable numero de veces. En realidad todo el tiempo estamos aplicando algoritmos para resolver problemas.

Un algoritmo es un conjunto de paso por paso, procedimientos o acciones que nos permiten alcanzar un resultado o resolver un problema.

Una receta en un libro de cocina seria un excelente ejemplo de algoritmo: la preparación de un platillo complicado se divide en pasos simples comprensibles para cualquier personas con experiencia en cocina.

Otro buen ejemplo de algoritmo podría ser la coreografía para un ballet clásico: se divide una danza complicada en una sucesión de pasos y posiciones básicas de ballet.

Muchas veces aplicamos el algoritmo de manera inadvertida, inconsciente o automática. Esto ocurre generalmente cuando el problema al que nos enfrentamos lo hemos resuelto con anterioridad un gran número de veces.

A continuación, se presenta un algoritmo simple para el proceso cotidiano de cambiar una llanta baja:

1. Aflojar los birlos de la llanta.
2. Levantar el carro con el gato.
3. Quitar los birlos de la llanta.
4. Quitar la llanta.
5. Poner la rueda de refacción.
6. Poner los birlos de la llanta.
7. Bajar el carro aflojando el gato.
8. Apretar los birlos de la llanta.

Las características que deben tener los algoritmos son las siguientes:

1. **Precisión:** Los pasos a seguir en el algoritmo se debe precisar claramente.
2. **Determinismo:** El algoritmo, dado un conjunto de datos de entrada idéntico, siempre debe arrojar los mismo resultados.
3. **Finitud:** El algoritmo, independientemente de la complejidad del mismo, siempre debe tener longitud finita.

Los algoritmos tienen comúnmente 3 secciones o módulos principales.

- Datos de Entrada.
- Procesamiento de los datos.
- Impresión de resultados.



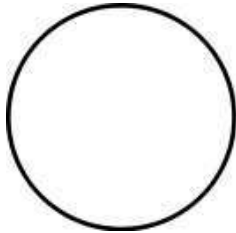


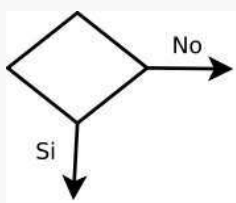
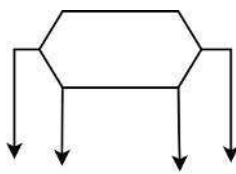

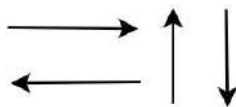
## Diagrama de flujo

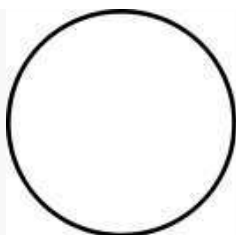
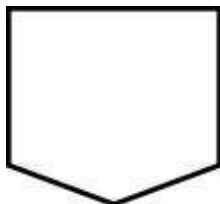

Un diagrama de flujo representa la esquematización gráfica de un algoritmo.

Realmente muestra gráficamente los pasos o procesos a seguir para alcanzar la solución de un problema. La construcción correcta del mismo es muy importante, ya que a partir de este se escribe el programa en un lenguaje de programación determinado. En este caso utilizaremos el lenguaje, aun que cabe recordar el diagrama de flujo se debe construir de

manera independiente al lenguaje de programación. El diagrama de flujo representa la solución del problema. El programa representa la implementación de un lenguaje de programación.

La tabla y las imágenes que veremos a continuación se utilizarán, junto con una explicación de las mismas imágenes, satisfaciendo las recomendaciones de la International Organization for Standardization (**ISO**), y el American National Standards Institute (**ANSI**).

Representación del Símbolo	Explicación del Símbolo
	Se utiliza para marcar el inicio y el fin del diagrama de flujo
	Se utiliza para introducir los datos de entrada, expresando lectura
	Representa un proceso. En su interior se colocan asignaciones, operaciones aritméticas, cambios de valor de variables
	Se utiliza para representar una decisión. En su interior se almacena una condición, y, dependiendo del resultado se sigue por una de las ramas o camino alternativos. Este símbolo se utiliza con pequeñas variaciones en las estructuras selectivas if e if-else que veremos en otros capítulos, así como en las estructuras repetitivas for, while y do-while
	Se utiliza para representar una decisión múltiple, switch, que analizaremos también mas adelante. En su interior se almacena un selector, y, dependiendo del valor de dicho selector, se sigue por una de las ramas o caminos alternativos
	Se utiliza para representar la impresión de un resultado, expresando escritura
	Expresan la dirección del flujo del diagrama

	Expresa conexión dentro de una misma pagina
	Representa conexión entre paginas diferentes
	Se utiliza para expresar un modulo de un problema o un subproblema, que hay que resolver antes de continuar con el flujo normal del diagrama

En la imagen siguiente se presentan los pasos que se deben seguir en la construcción de un diagrama de flujo. El procesamiento de los datos generalmente relacionado con el proceso de toma de decisiones. Además, es muy común repetir un conjunto de pasos.



## Reglas para la construcción de diagramas de flujo

Los símbolos presentados, colocados en los lugares adecuados, permiten crear una estructura gráfica flexible que ilustra los pasos a seguir para alcanzar un resultado específico. El diagrama de flujo facilita entonces la escritura del programa en C. Lo que veremos a continuación se presenta el conjunto de reglas para la construcción de diagramas de flujo:

1. Todo diagrama de flujo debe tener un inicio y un fin.
2. Las líneas utilizadas para indicar la dirección del flujo del diagrama deben ser rectas: ya sea verticales u horizontales.

3. Todas las líneas utilizadas para indicar la dirección del flujo del diagrama deben estar conectadas. La conexión puede ser un símbolo que exprese lectura, proceso, decisión, impresión, conexión o fin del diagrama.
4. El diagrama de flujo debe construirse de arriba hacia abajo y de izquierda a derecha.
5. La notación utilizada en el diagrama de flujo debe ser independiente del lenguaje de programación. La solución presentada se puede escribir posteriormente en diferentes lenguajes de programación.
6. Al realizar una tarea compleja, es conveniente poner comentarios que expresen o ayuden a entender lo que hayamos hecho.
7. Si la construcción del diagrama de flujo requiriera más de una hoja, debemos utilizar los conectores adecuados y enumerar las páginas correspondientes.
8. No puede llegar más de una línea a un símbolo determinado.

## Desventajas de los diagramas de flujo

Aun que me parecen muy interesantes como primer acercamiento a la programación ya que son fáciles de entender. De hecho se utilizan fuera de la programación como esquema para ilustrar el funcionamiento de algoritmos sencillos.

Sin embargo cuando el algoritmo se complica, el diagrama de flujo se convierte en ininteligible. Además los diagramas de flujo no facilitan el aprendizaje de la programación estructurada.

## Pseudocódigo

Las bases de la programación estructurada fueron enunciadas por **Niklaus Wirth**. Según este científico cualquier problema algorítmico podía resolverse con el uso de estos 3 tipos de instrucciones:

1. **Secuenciales** Instrucciones que se ejecutan en orden normal. El flujo del programa ejecuta la instrucción y pasa a ejecutar la siguiente.
2. **Alternativas** Instrucciones en las que se evalúa una condición y dependiendo si el resultado es verdadero o no, el flujo del programa se dirigirá a una instrucción u otra.
3. **Iterativas** Instrucciones que se repiten continuamente hasta que se cumple una determinada condición.

El tiempo le ha dado completamente la razón y ha generado una programación que insta a todo programador a utilizar solo instrucciones de esos 3 tipos. Es lo que se conoce como programación estructurada.

Pero el propio **Niklaus Wirth** diseñó el lenguaje pascal como el primer lenguaje estructurado. Lo malo es que pascal al ser un lenguaje completo incluye instrucciones excesivamente orientadas al ordenador.

Por lo que en aquella época se aconsejó para el diseño de algoritmos estructurados el uso de un lenguaje especial llamado pseudocódigo, que además de puede traducir a cualquier idioma y lenguaje de programación.

El pseudocódigo además permite el diseño modular de programas y el diseño descendente gracias a esta posibilidad.

Debes entender además de que existe un gran número de tipos de pseudocódigos, es decir no hay un pseudocódigo 100% estándar. Pero si hay una gran cantidad de detalles aceptados por todos los que escriben pseudocódigos. Aquí te comentare el pseudocódigo aceptado en español. Hay que tomar en cuenta que muchas de las ideas de pseudocódigo se basa en pascal, por lo que la traducción es casi directa.

El pseudocódigo son instrucciones escritas en un pseudolenguaje orientado a ser entendido por una persona no por un ordenador. Por ello en pseudocódigo solo se pueden utilizar ciertas instrucciones.

Entre las instrucciones que se pueden utilizar están:

- **De Entrada/Salida.** Para leer o escribir datos desde el programa hacia el usuario.
- **De Proceso.** Operaciones que realiza el algoritmo (Suma, Resta, Multiplicación, División, cambio de valor, etc.)
- **De Control de Flujo.** Instrucciones alternativas o iterativas (bucles y condiciones).
- **De Declaraciones.** Creación de variables y subprogramas.
- **Llamadas a Subprogramas.**
- **Comentarios.** Notas que se escriben junto al pseudocódigo para explicar mejor su funcionamiento.



# Metodologías de la programación

Lo mas común con lo que nos topamos cada vez que aprendemos un lenguaje de programación, es querer iniciar inmediatamente a programar sin antes haber pensado un poco sobre el problema a resolver. Y al toparnos con problemas de diseño solemos recurrir a rehacer las cosas. Esta parte es para tener cuando menos una buena idea desde donde iniciar y como hacerlo.

## Introducción

Se entiende por metodología el conjunto de reglas y pasos estrictos que se siguen para desarrollar una aplicación. Hay diversas metodologías, independientemente de la vayamos a utilizar suele haber una serie de pasos comunes en todas ellas (*Relacionado con el ciclo de vida de la aplicación*):

1. Análisis.
2. Diseño.
3. Codificación.
4. Ejecución.
5. Prueba.
6. Mantenimiento.

## Análisis

Al programar aplicaciones siempre se debe realizar un cierto análisis.

El análisis estudia los requisitos que ha de cumplir la aplicación.

El resultado de esto es una hoja de especificaciones en la que aparecen los requerimientos de la aplicación. Esta hoja es redactada por el o la analista o la persona responsable del proceso de creación de la aplicación.

En la creación de algoritmos sencillos, el análisis consistiría unicamente en:

- **Determinar las entradas.** Es decir, los datos que posee el algoritmo cuando comienza su ejecución. Esos datos permiten obtener el resultado.
- **Determinar las salidas.** Es decir, los datos que obtiene el algoritmo como resultado. Lo que que algoritmo devuelve al usuario.
- **Determinar el proceso.** Se estudia cual es el proceso que hay que realizar.

## Diseño

En esta fase se crean esquemas simbolizan la aplicación. Estos esquemas los elaboran analistas. Gracias a estos esquemas se simboliza la aplicación. Estos esquemas en definitiva se convierten en la documentación fundamental para plasmar en papel lo que el programador debe hacer.

En estos esquemas se puede simbolizar: la organización de los datos de la aplicación, el orden de los procesos que tiene que realizar la aplicación, la estructura física (*en cuanto archivo y carpetas*) que utilizara la aplicación.

La creación de estos esquemas se puede hacer en papel, o utilizar una herramienta para hacerlo.

En el caso de la creación de algoritmos, conviene en esta fase usar el llamado diseño descendente. Mediante este diseño el problema se divide en módulos, que a su vez, se vuelven a dividir a fin de solucionar problemas mas concretos. Al diseño descendente se le suele llamar **Top-Down**. Gracias a esta técnica un problema complicado se divide en pequeños problemas que son mas fácilmente solucionables.

Siempre existe en el diseño la zona principal que es el programa principal que se ejecutara cuando el programa este codificado en un lenguaje de programación.

En la construcción de aplicaciones complejas en esta fase se utilizan gran cantidad de esquemas para describir la organización de los datos y los procedimientos que ha de seguir el programa. En pequeños algoritmos se utilizan esquemas mas sencillos.

## Codificación

Es la escritura de la aplicación en un lenguaje de programación. Normalmente la herramienta utilizada en el diseño debe ser compatible con el lenguaje que se utilizara. Es decir si se utiliza un lenguaje orientado a objetos, la herramienta de diseño debe ser una herramienta que permita utilizar objetos.

Pero no comas ansias, se muy bien que debes estar ya con muchas ganas de empezar a programar en **C** pero esta parte es de suma importancia y te dará las armas suficientes para hacer algo mas grande sin que tengas que hacer re-estructuraciones al momento de tu codificación.

## Ejecución

Tras la escritura del código, mediante un compilador o un intérprete podemos traducir el código a uno interpretable por la máquina. En este proceso pueden detectarse errores en el código que impiden su transformación. En este caso el software encargado de la traducción avisa de esos errores para que el programador los pueda corregir.

## Pruebas

Las pruebas o verificaciones de un programa es el proceso de ejecución del programa con una amplia variedad de datos de entrada, que determinarán si el programa tiene errores (*bugs*). Para realizar las verificaciones se debe desarrollar una amplia gama de datos de test, valores normales de entrada, valores extremos de entrada que comprueben los límites del programa y valores de entrada que comprueben aspectos especiales del programa.

La *depuración* es el proceso de encontrar los errores del programa y corregir o eliminar dichos errores. Cuando se ejecuta un programa, comúnmente se suele producir 3 tipos de errores:

1. **Errores de Compilación.** Se producen normalmente por uso incorrecto de las reglas del lenguaje de programación y suelen ser *errores de sintaxis*. Si existe un *error de sintaxis*, la computadora no puede comprender la instrucción, no se obtendrá el programa objeto y el compilador imprimirá una lista de todos los errores encontrados durante la compilación.
2. **Errores de Ejecución.** Estos se producen por instrucciones que la computadora puede comprender pero no ejecutar. Ejemplos típicos son: división por cero y raíces cuadradas de números negativos. En estos casos se detiene una ejecución del programa y se imprime un mensaje de error.
3. **Errores lógicos.** Como su nombre lo dice se producen en la lógica del programa y la fuente del error suele ser el diseño del algoritmo. Estos errores son los más difíciles de detectar, ya que el programa puede funcionar y no producir errores de compilación ni de ejecución, y solo puede advertirse el error por la obtención de resultados incorrectos. En este caso se debe volver a la fase de diseño del algoritmo, modificar el algoritmo, cambiar el programa fuente, compilar y ejecutar una vez más. De ahí que la importancia de las primeras fases es crucial.

## Mantenimiento

En esta fase se crea la documentación del programa (Aun que no lo creas es un paso fundamental para la creación de aplicaciones). Gracias a esa documentación se puede corregir futuros errores o renovar más fácilmente ciertas partes.

La documentación de un problema consta de las descripciones de los pasos a dar en el proceso de resolución de dicho problema. La importancia de la documentación debe ser destacada por su decisiva influencia en el producto final. Programas pobremente documentados son difíciles de leer, mas difíciles de depurar y casi imposibles de mantener o modificar.

La documentación de un programa puede ser *interna* y *externa*. La **documentación interna** es la contenida en líneas de comentarios. La **documentación externa** incluye análisis, diagramas de flujo y/o pseudocódigo, manuales de usuario con instrucciones para ejecutar el programa y para interpretar los resultados.

La documentación es vital cuando se desea corregir posibles errores futuros o bien cambiar el programa. Tales cambios se denominan *mantenimiento del programa*. Después de cada cambio de documentación debe ser actualizada para facilitar cambios posteriores.

## Programación Modular

Es uno de los métodos de diseño mas flexibles y potentes para mejorar la productividad de un programa. En la programación modular el programa se divide en módulos (partes independientes), cada uno de los cuales ejecutan una única actividad o tarea y se codifican independientemente de otros módulos. Cada uno de estos módulos se analiza, codifica y pone punto por separado. Cada programa contiene un modulo denominado **programa principal** que controla todo lo que sucede transfiriendo el control a submodulos (también llamados *subprogramas*), de modo que ellos puedan ejecutar sus funciones; sin embargo, cada submodulo devuelve el control al modulo principal cuando haya completado su tarea. Si la tarea asignada a cada submodulo es demasiado compleja, este deberá romperse en otro módulos mas pequeños. El proceso sucesivo de subdivisión de módulos continua hasta que cada modulo tenga solamente una tarea especifica que ejecutar.

Esta tarea puede ser *entrada, salida, manipulación de datos, control de otros módulos o alguna combinación de estos*. Un módulo puede transferir temporalmente (bifurcar) el control a otro modulo; sin embargo, cada modulo debe eventualmente devolver el control al modulo del cual se recibe originalmente el control.

Los módulos son independientes en el sentido en que ningún modulo puede tener acceso directo a cualquier otro modulo excepto al modulo al que llama y sus propios submodulos. Sin embargo, los resultados producidos por un modulo pueden ser utilizados por cualquier otro modulo cuando se transfiera a ellos el control.

Dado que los módulos son independientes, diferentes programadores pueden trabajar simultáneamente en diferentes partes del mismo programa. Esto reducirá el tiempo del diseño del algoritmo y posterior codificación del programa. Además, un modulo se puede

modificar radicalmente sin afectar a otros módulos, incluso sin alterar su función principal.

La descomposición de un programa en módulos independientes mas simples se conoce también como método de **Divide y vencerás**. Por lo que se diseña cada modulo con independencia de los demás, y siguiendo el método ascendente o descendente se llegara hasta la descomposición final del problema en módulos en forma jerárquica.

## Programación estructurada

Según los términos de la *programación modular*, *programación descendente* y *programación estructurada* se introdujeron en la segunda mitad de la década de los sesenta y a menudo se utilizan son sinónimos aunque no significan lo mismo. La programación modular y descendente ya se ha examinado anteriormente. La programación estructurada significa escribir un programa de acuerdo a las siguientes reglas:

- El programa tiene un diseño modular.
- Los módulos son diseñados de modo descendente.
- Cada modulo se codifica utilizando las tres estructuras de control básicas: secuencia, selección y repetición.

El termino **programación estructurada** se refiere a un conjunto de técnicas que han ido evolucionando desde los primeros trabajo de *Edgar Dijkstra*. Estas técnicas aumentan considerablemente la productividad del programa reduciendo en elevado grado el tiempo requerido para escribir, verificar, depurar y mantener los programas. Ademas utiliza un numero limitado de estructuras de control que minimizan la complejidad de los programas y, por consiguiente, reducen los errores; hace los programas mas fáciles de escribir, verificar, leer y mantener. Los programas deben estar dotados de una estructura.

La programación estructurada es el conjunto de técnicas que incorporan:

- Recurso abstractos.
- Diseño descendente.
- Estructuras básicas.

## Recurso Abstractos

La programación estructurada se auxilia de los recursos abstractos en lugar de los recursos concretos que dispone un determinado lenguaje de programación. Descomponer un programa en términos de recursos abstractos -Según Dijkstra- consiste en descomponer una determinada acción compleja en términos de un numero de acciones mas simples capaces de ejecutarlas o que constituyan instrucciones de computadoras disponibles.

## Diseño descendente (top-down)

Es el proceso mediante el cual un problema se descompone en una serie de niveles o pasos sucesivos de refinamiento. La metodología descendiente consiste en efectuar una relación entre las sucesivas etapas de estructuración de modo que se relacionen unas con otras mediante entradas y salidas de información. Es decir, se descompone el problema en etapas o estructuras jerárquicas, de forma que se puede considerar cada estructura desde dos puntos de vista:

- ¿Qué hace?
- ¿Cómo lo hace?

## Estructuras de control

Las *estructuras de control* de un lenguaje de programación son métodos de especificar el orden en que las instrucciones de un algoritmo se ejecutaran. El orden de ejecución de las sentencias o instrucciones determina el *flujo de control*. Estas estructuras de control son, por consiguiente fundamentales en los lenguajes de programación y en los diseños de algoritmos, especialmente los pseudocódigos.

Las tres estructuras de control básicos son:

- **Secuencia.**
- **Selección.**
- **Repetición.**

## Creación de Algoritmos

Independientemente de la notación que utilicemos para escribir algoritmos, estos contiene instrucciones a realizar por el ordenador. Lógicamente la escritura de estas instrucciones siguen unas normas muy estrictas. Las instrucciones pueden ser de estos tipos:

- **Primitivas:** Son acciones sobre los datos del programa, estas pueden ser:
  - Asignación.
  - Instrucciones de Entrada/Salida.
- **Declaraciones:** Sirven para advertir y documentar el uso de variables y subprogramas en el algoritmo.
- **Control:** Sirven para alterar el orden de ejecución del algoritmo. En general el algoritmo se ejecuta secuencialmente. Gracias a estas instrucciones el flujo del algoritmo depende de ciertas condiciones que nosotros mismos indicamos.



## Editores

Como lo dije casi al principio, el editor que desees usar para desarrollar es irrelevante, lo que pretendo ahora es mostrarte con gran detalle y detenimiento las bondades de mi editor favorito por lo que iniciaremos con su nombre:



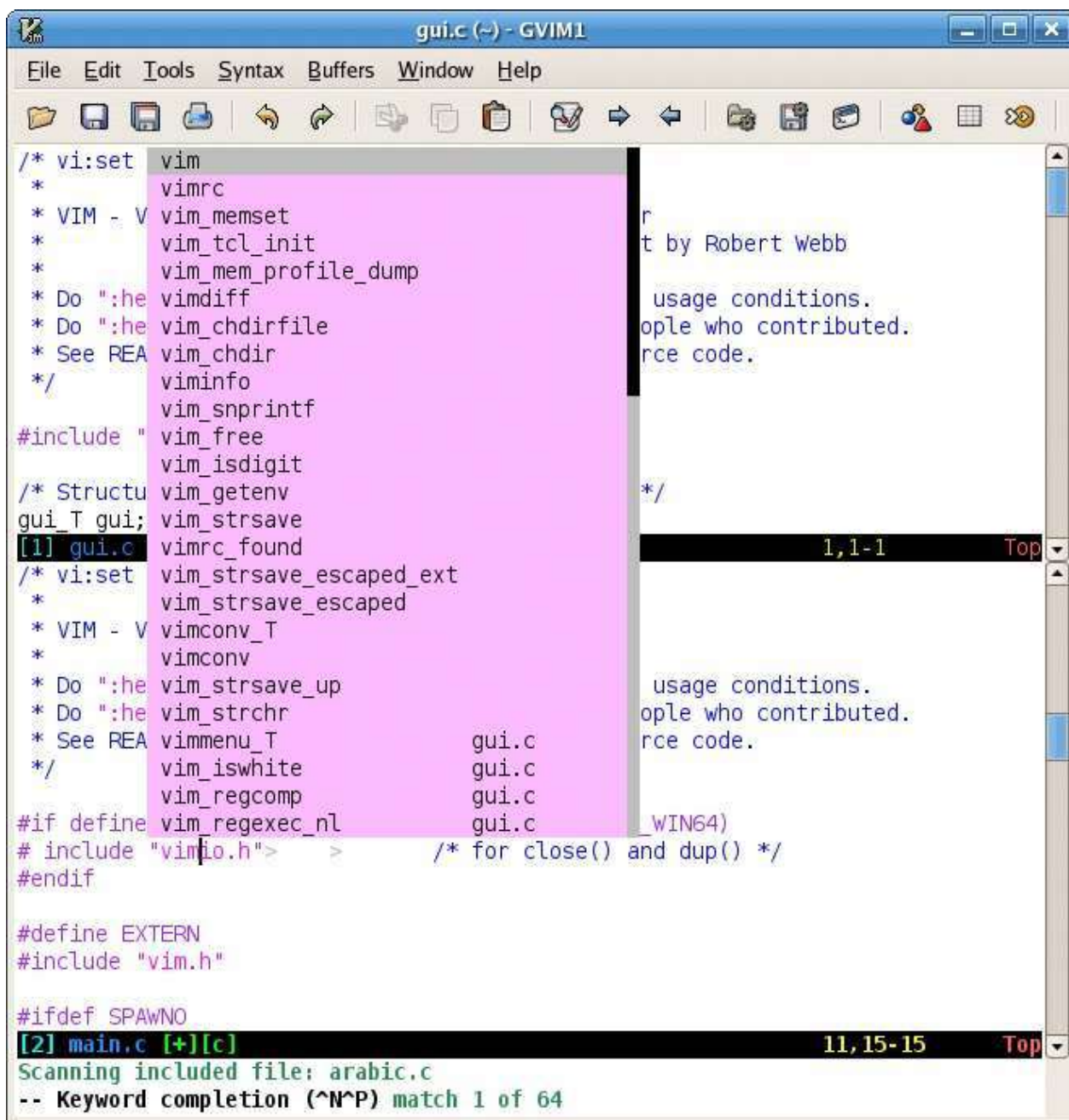
Se muy bien que existen varios editores pero este en especial me ha parecido muy sencillo y poderoso, ademas me encuentro trabajando en algo que contare mas adelante y te facilitara en gran medida la vida si es que quieres programar usando este editor.

## Vim

Como ya dijimos **Vim** o ("**VI IM**proved") es un clon de **Vi**, es decir, un programa similar al editor de textos "**Vi**".



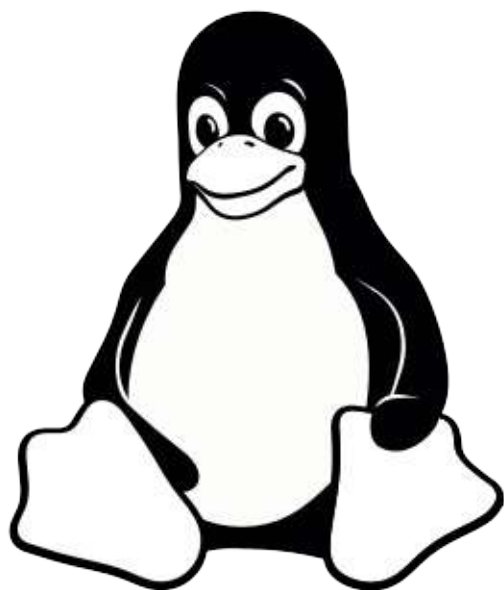
**Vim** no solo trabaja en modo de texto en cualquier terminal, sino que también tiene un interfaz gráfica para el usuario, es decir, menús y soporte para el ratón.



Está disponible para muchas plataformas y tiene muchas características añadidas en comparación con **Vi**. Es compatible con casi todos los comandos de **Vi** - excepto con los errores.

Esta disponible en un gran numero de sistemas operativos entre ellos: Linux:

- *Linux:*



- OSx:



- *Windows:*



- Si eres usuario de sistemas *BSD* debo informarte que también lo vas a poder usar:



freeBSD

El copyright está en las manos del autor principal y mantenedor, **Bram Moolenaar** [bram@vim.org](mailto:bram@vim.org). Vim es un "**programa-de-caridad**" "**charity-ware**"), es decir que se sugiere que hagas una donación y en especial a una asociación de huérfanos en Uganda.

Y si te lo preguntabas vim a pesar de entrar en **charity-ware** es software *Open Source* y todos los que quieran apoyar son bienvenidos para contribuir en algo al proyecto.

## Características

Es un editor para un Principiante - Amigable para el usuario:

Es mucho más fácil para los principiantes que VI debido a la disponibilidad de ayuda en línea bien extensa, comandos para "deshacer" (undo) y "rehacer" (redo) (¡no te preocupes mucho con los errores - simplemente usa undo y redo!), soporte para el ratón y también iconos y menús configurables (GUI).

Soporta la edición de derecha-a-izquierda (ej. con el **Árabe**, **Farsi**, **Hebreo**), y textos en **multi-octeto**, es decir, lenguajes con caracteres gráficos representados por más de un "octeto", por ejemplo **Chino**, **Japonés**, **Coreano (Hangul)**, técnicamente hablando, Vim soporta texto escrito en **UTF-8** y **Unicode**.

```
Lo genial que es vim:
Ruso:
    привет мир
Español:
    Hola mundo.
Ingles:
    Hello World.
Frances:
    Bonjour tout le monde
Portuges:
    Olá, mundo
Japones:
    こんにちは
Polaco:
    Witaj świecie
Tailandes:
    สวัสดีชาวโลก
Aleman:
    Hallo Welt
Chino:
    你好世界
```

Vamos a poder seleccionar el texto "**visualmente**" antes de que "*operen*" en él, ej. copiar, remover, substituir, mover la posición a la izquierda o derecha, cambiar la capitalización de las letras o el formato del texto incluso preservando la indentación del mismo. También la selección y operaciones en bloques de texto rectangulares.

**Vim** tiene comandos que completan su entrada de información sea con comandos, nombres de fichero, o palabras.

Del mismo modo tiene "*autocommands*" para la ejecución automática de los comandos (ej. Descompresión automática de ficheros comprimidos). Reconoce automáticamente el tipo de ficheros (**DOS**, **mac**, **Unix**) y también le permite el guardar el archivo en cualquier otro formato ¡no hay necesidad de usar `unix2dos` para usar en Windows nunca más!

Tiene un "*historia*" para los comandos y las búsquedas, así que puedes llamar nuevamente los comandos o el patrón de búsqueda anteriores para editarlos.

Permite "*grabar*" una serie de acciones de edición para poder ejecutarlas nuevamente cuando se realizan tareas repetitivas.

Tiene límites de memoria mucho más grandes para la longitud de línea y el tamaño del almacenador intermediario (buffer) en comparación con **VI** normal.

Permite corregir de múltiples almacenadores intermediarios y puedes partir la pantalla en muchas *sub-ventanas* (**horizontal** o **verticalmente**), así que vas a poder ver muchos ficheros o muchas partes de algunos ficheros.

Tiene un lenguaje de escritura incorporado para poder extenderlo fácilmente.

Permite el usar desplazamientos relativos para los comandos de búsqueda, así que se puede poner el cursor inmediatamente en el lugar *después* del texto encontrado.

Permite para salvar la información de una sesión de edición en un fichero ("*viminfo*") lo cual permite que sean usados en una subsecuente sesión de edición, ej. la lista de almacenadores intermediarios, de las marcas de fichero, de los registros, comandos y de la historia de las búsquedas.

Puede expandir las tabulaciones dentro del texto usando caracteres de espacio (**expandtab**, **retab\*** ).

Permite el encontrar texto en ficheros usando un índice con las "*etiquetas*" (**tags** ) junto con muchos otros comandos que manipulan la lista de etiquetas.

Muestra el texto en color según su "*lenguaje de programación*". Tu mismo puedes definir el "**lenguaje**" ("**sintaxis**") de los ficheros. Viene con muchísimos ficheros de sintaxis para la coloración del texto en los lenguajes de programación comunes (**Ada**, **C**, **C++**, **Eiffel**, **FORTRAN**, **Haskell**, **Java**, **lisp**, **Modula**, **PASCAL**, **prolog\***, **Python**, **scheme**, **SQL**, **Verilog**, **VisualBasic**, **Ensamblador**, **Ruby**, **Javascript**), programas de matemáticas (**arce**, **Matlab**, **Mathematica**, **SAS**), texto que use marcado específico (**DocBook**, **HTML**, **LaTeX**, **PostScript**, **SGML-LinuxDoc**, **TeX**, **WML**, **XML**, **Markdown**, **Json**), retornos de programas (**diff**), ficheros de la configuración de programas (**Apache**, **autoconfig**, **BibTeX**, **CSS**, **CVS**, **IDL**, **LILO**, **mail**, **samba**, **Grub**, **Nginx**), lenguajes de escritura del procesador de comandos (**Shell**) y de configuración (shells: **sh**, **csch**, **ksh**, **zsh**), lenguajes de procesamiento de texto (**awk**, **gawk**, **Perl**, **sed**, **yacc**), ficheros de sistema (*printcap*, *Xdefaults*) y por supuesto para **Vim** y sus textos de ayuda.

Ademas de que tiene otras extensiones como snippets, o verificadores de sintaxis que lo hacen mucho mas potente y mas cercano a un **IDE**, de ahí que les este dando una pequeña entrada sobre lo extraordinario que me resulta este editor.

En este momento me encuentro trabajando en algunas extensiones precisamente con este editor para hacerlo mas robusto y poder usarlo mas como un **IDE** para los lenguajes **C**, **C++**, **LaTeX**, **Python**, **Bash**, **Java**, **Javascript**, **Web** entre otros. Al mismo tiempo estoy recolectando algunos scripts que me resalten sintaxis en varias librerías entre ellas esta **Gtk**, **Gdk**, **MPI**, **OpenGL**, **ANSI-C**, me encuentro trabajando en la actualización de **Gtk**, **Gdk**, **Ncurses** y la de **Linux Kernel-API**.

# Iniciando con vim

Ya vimos por que **vim** es un editor muy potente y que ademas dispone de muchos mandatos, demasiados para ser explicados en este libro. Esta parte está dedicada para describir suficientes mandatos para que seas capaz de aprender fácilmente a usar **vim** como un editor de propósito general.

Es importante recordar que esta parte es pensada para enseñar con la práctica. Esto significa que es necesario ejecutar los mandatos para aprenderlos adecuadamente. Si únicamente se lee el texto, se olvidarán los mandatos.

Para poder iniciar a usar el editor primero deberemos descargarlo para esto les dejare los siguiente enlaces:

1. [Pagina Principal](#)
2. [Preguntas Frecuentes](#)
3. [Código](#)

Como siempre mi recomendación sera la siguiente:

- Compilar unicamente si cumples con las dependencias de construcción y sabes lo que haces.
- De otro modo te recomiendo instalar **vim** con el gestor de paquetes de la distribución que estés ocupando:

Para **Fedora** deberás usar el siguiente comando:

```
dnf install vim
```

Para **Ubuntu** o **Debian** deberás usar el siguiente comando:

```
apt install vim ó apt-get install vim
```

**Nota:** Hay un reporte sobre que no funcionan algunos plugins de vim para ello tendremos que instalar los siguiente:

```
apt install vim-nox ó apt-get install vim-nox
```

Para **ArchLinux** deberás usar el comando siguiente:

```
pacman -S vim
```

Para **Suse** deberás usar el comando siguiente:

```
zypper install vim
```

Una vez realizado todo esto seguiremos viendo como funciona **vim**.

La ultima versión de **vim** es la **7.4.1990** hasta Julio 2016.

Para iniciar a ocupar el editor tendrás que hacerlo desde una terminal ejecutando lo siguiente:

```
$ vim <archivo>
```

De manera básica vim tiene varios modos entre ellos.

- **Normal**
- **Visual**
- **Insertar**
- *Entre algunos mas.*

Cuando entras a **vim** inicias el modo normal:

A screenshot of the Vim editor's startup screen. The background is dark gray. The text is white and green. It displays the Vim logo, version information (7.4.1830), and credits to Bram Moolenaar. It also mentions that Vim is open-source code and can be distributed freely. There are instructions on how to get help, exit, or see the version. At the bottom right, it shows the status bar with '0,0-1' and 'Todo'.

Pero al teclear algunas letras puedes acceder a algunos modos, aun que te parezca muy difícil al comienzo veras que te vas a acostumbrar muy pronto.

Para entrar al modo edición tendrás que teclear la letra **"i"**.



Esto mismo inmediatamente te cambiara de modo para que puedas iniciar la edición del documento.

Y para entrar al modo visual tendrás que teclear la letra **"v"**.



Esto mismo inmediatamente te cambiara de modo para que puedas seleccionar algunas parte de tu texto a tu gusto.

## Iniciando.

Ahora seguramente ya habrás incursionado en **vim** pero no sabes ni como salir ni guardar tus cambios. Para eso deberas pulsa la tecla "**Esc**" para asegurarte de que estas en modo **Normal**:



Para salir de archivo sin guardar nada deberás pulsar las teclas: **:q!**



Para salir del archivo guardando los cambios deberás pulsar las teclas: **:wq**



Si lo que quieres es solo guardar tu archivo sin salir lo lograras pulsando las siguiente



teclas: **:w**

## Moviendo el cursor.

Para mover el cursor, pulse las teclas **H,J,K,L** de la forma que se indica.





Como puede verse en la imagen la tecla **H** está a la izquierda y mueve el cursor a la izquierda, la tecla **L** está a la derecha y mueve el cursor a la derecha, la tecla **J** parece una flecha que apunta hacia abajo y mueve el cursor abajo por ultimo la tecla **K** parece una flecha que apunta hacia arriba y mueve el cursor arriba.

## Agregar texto al archivo.

Este editor es considerado como un editor modal, esto significa que puede tener diferentes comportamientos dependiendo del modo en el que nos encontremos y como vimos antes los 2 modos básicos son: **modo normal** y **Modo insertar**. Desde el primer momento en el que ingresamos al editor nos topamos con el **modo normal**. Para ocupar el **modo inserción** teclearemos la letra "i":



Y es así como ya podremos ingresar texto en nuestro archivo, ahora podrás iniciar la edición sin problemas hasta el momento en el que tecles :



## Borrando caracteres.

Existen muchas combinaciones para realizar el borrado entre las mas usuales se encuentra las opciones "**x**", "**dw**", "**dd**", "**d\$**", "**d#**" y "**de**".

Donde:



- **x** Elimina un solo caracter.
- **dw** Elimina desde el cursor hasta el final de la palabra, incluyendo el espacio.



- **dd** Elimina toda una linea entera.
- **d\$** Elimina todos los caracteres desde donde se encuentra posicionado el cursor.



- **d#** Elimina los caracteres de derecha a izquierda hasta donde se encuentra



posicionado el cursor.

# Relación computadoras y los lenguajes de programación.

Esta parte es una sugerencia de varias personas donde piden que se explique un poco más a detalle como se relacionan las computadoras con los lenguajes de programación, lo quise omitir pensando que eso lo podrían buscar en algunos otros libros pero creo que no estaría completo el libro sin esta parte. Las computadoras hoy en día se han convertido en una herramienta esencial en muchas áreas: industria, gobierno, ciencia, educación, etc. El papel de los programas y aplicaciones juega un papel crucial; sin una lista de instrucciones a seguir, la computadora es virtualmente inútil. Y los lenguajes de programación nos permiten escribir esos programas y, por consiguiente, comunicarnos con las computadoras a través de ellos. Una computadora procesa los datos y los convierte en información significativa. Para conseguir esos resultados, un programador necesita conocimientos tanto de **hardware** como de **software**.

## Organización de una computadora.

Desde la década de los 40's se crearon las primeras computadoras con nombres muy extravagantes como **Atanasoff-Berry**, **UNIVAC**, **ENIAC** o **EDVAC**. La aparición de **IBM** fue el punto de partida de la moderna computación, pasando de computadoras imposibles de pagar por cualquier persona a unas un poco más accesibles. Del mismo modo el software se vio cambiado desde el inicio de una compañía llamada **Microsoft** que fue en gran medida una de las que propició la creación del **Software Libre** y el **Open Source**. Aunque a primera vista puedan parecer sinónimos los **datos** e **información** existe una gran diferencia entre ellos.

Los **datos** consta de echos en bruto.

La **información**, por otra parte, son datos procesados.

La información tiene datos significativos; los datos en su forma original (en bruto) no lo son tanto. La computadora procesa esos datos y la información significativa es el resultado que se conoce muy comúnmente como *salida*. Los datos, por si mismo no sirven de mucho y no son muy útiles para las personas que los manipulan y necesitan tomar decisiones con ellos.

Los programas modernos producen información en muchos formatos. Estos programas reproducen música, se comunican con otras computadoras a través de la red y más aun con los smart phones que en los últimos años se han convertido en algo casi obligatorio.

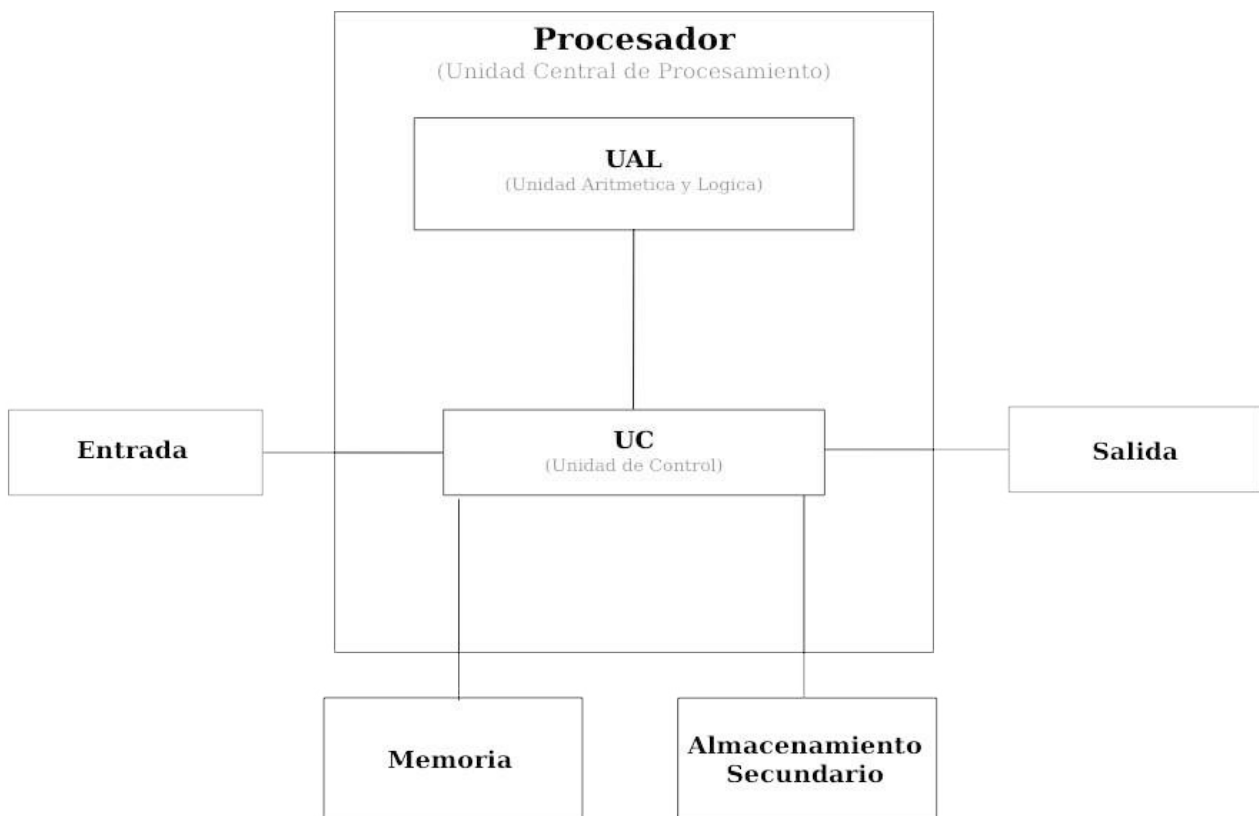
Todos estos envían una salida a la pantalla, impresora, es decir los datos de entrada y los datos de salida pueden ser, realmente, cualquier cosa, texto, imágenes, música.

Una computadora necesita tanto del **hardware** como del **software** para poder ser usada real y prácticamente. El **hardware** de la computadora sin el software que le acompaña es como si tuviéramos un libro con paginas en blanco. La portada, contraportada y las paginas interiores constituyen el **hardware** del libro, pero el libro no es útil sin ningún texto, el **software**.

## Hardware

Cuando un usuario interactúa con una computadora, proporciona una entrada; en respuesta, la computadora procesa la entrada devolviendo una salida valiosa al usuario. La entrada puede ser en formato de ordenes o instrucciones dadas, texto, números o imágenes. La salida puede ser el resultado: cálculos en *calc* de **libreoffice**, una oficio en *writer* o un auto moviéndose por la pantalla en un juego como **super tux**. Una computadora necesita disponer de un conjunto de funcionalidades y proporcionar la capacidad de:

1. Aceptar la entrada.
2. Visualiza o presentar la salida.
3. Almacenar la información en un formato consistente.
4. Ejecutar operaciones aritméticas o lógicas bien sobre datos de entrada o bien sobre datos de salida.
5. Monitorizar, controlar y dirigir las operaciones globales y de secuencia del sistema.



## Procesador

El procesador es el dispositivo interior de la computadora que ejecuta las instrucciones del programa o aplicación. También se le suele conocer como **UCP** (*Unidad Central de Procesamiento*). En el mercado existen muchas marcas disponibles para computadoras de escritorio o laptop como:

1. **INTEL**
2. **AMD**

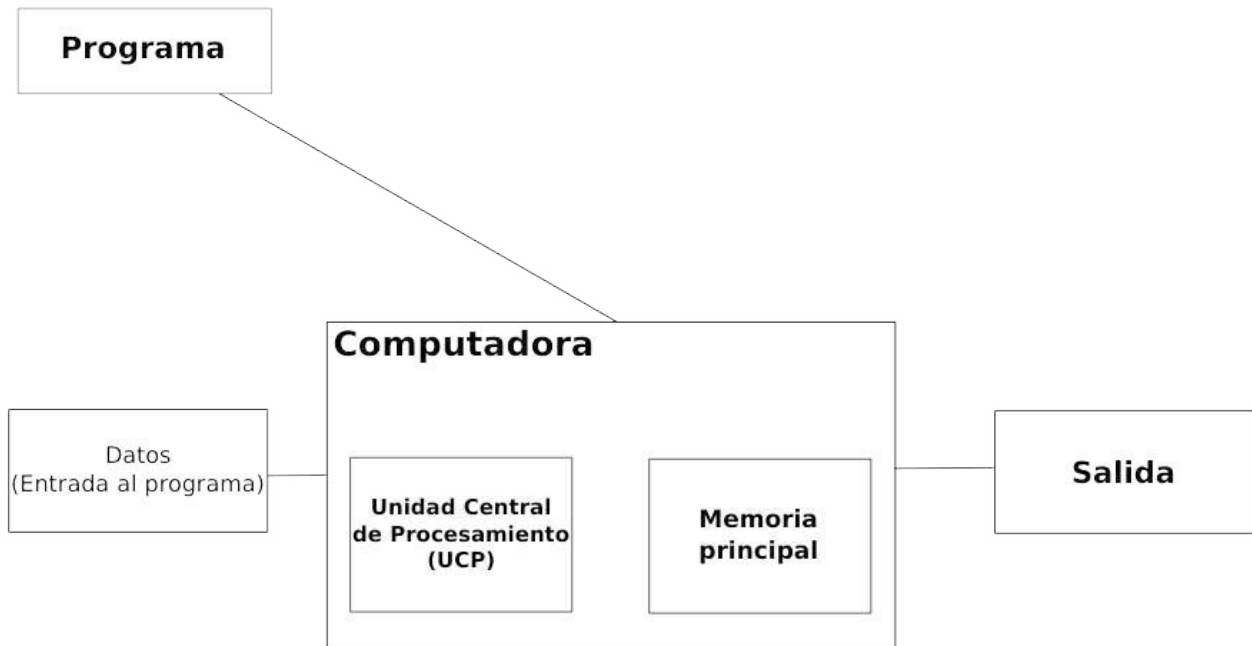
O para móviles como:

1. **ARM**
2. **INTEL**
3. **AMD**
4. **NVIDIA**
5. **SAMSUNG**
6. **QUALCOMM**

Ya casi no existe persona que no tenga un smart phone y ha sido este el caso en el que mucha gente ahora prefiere desarrollar apps que aprender el bajo nivel, sin embargo ignoran que lo que sustenta esas apps son el sistema operativo.

El procesador solo puede ejecutar instrucciones simples, tales como cálculos aritméticos sencillos o desplazamientos de número por diferentes posiciones. Sin embargo, la velocidad a la cual se realizan estos cálculos debe ser muy grande y esta característica le permite ejecutar instrucciones que realizan cálculos complejos.

La **UCP**, dirige y controla el proceso de información realizado por la computadora, procesando y manipulando la información almacenada en la memoria. Se puede recuperar esa información de la memoria. También puede almacenar los resultados de estos procesos en memoria para su uso posterior.



La **UCP** consta de dos componentes:

1. **Unidad de Control (UC)**
2. **Unidad Aritmético Lógica (UAL)**

La **Unidad de Control (UC)** coordina las actividades de la computadora, determinando que operaciones se deben realizar y en que orden; así mismo controla y sincroniza todo el proceso de la computadora.

La **Unidad Aritmético Lógica (UAL)** Realiza operaciones aritméticas y lógicas, tales como suma, resta, multiplicación, división y comparaciones. Los datos en la memoria central se pueden leer o escribir por la **UCP**.

## Memoria

Otra parte muy importante de una computadora es la memoria. La unidad de memoria almacena la información en un formato lógicamente consistente. Normalmente, tanto las instrucciones como los datos se almacenan en memoria, con frecuencia en áreas distintas y separadas. La memoria se divide en dos categorías:

1. **Memoria Principal**
2. **Memoria Auxiliar**

La **memoria Principal** contiene el programa en ejecución y los resultados de los cálculos intermedios de la computadora. Se le suele conocer mas como **Memoria Ram**. La **Memoria Auxiliar** o almacenamiento secundario consta de dispositivos utilizados para almacenar los datos en modo permanente. Cuando se necesitan los datos se pueden recuperar de estos dispositivos.

El programa se almacena en memoria externa de modo permanente pero cuando se ha de ejecutar debe transferirse a la memoria central. Este proceso se realiza mediante ordenes al sistema operativo que realiza las operaciones correspondientes.

## Microprocesador

Es un circuito integrado que controla y realiza las funciones y operaciones con los datos. En realidad el microprocesador representa la Unidad Central de Procesamiento o procesador, y popularmente cuando se habla de una computadora el termino que se utiliza en las características técnicas para referirse a la Unidad Central de Procesamiento. Las velocidades de un microprocesador se mide en **MegaHercios (MHz)** aunque es mas común que encuentres los **GigaHercios (Ghz)**.

INTEL	AMD
<b>Pentium Celeron</b> ( <i>Sexta Generación</i> )	<b>APU Athlon</b> ( <i>Cuarta Generación</i> )
<b>Core i3</b> ( <i>Sexta Generación</i> )	<b>APU ATHLON</b> ( <i>Cuarta Generación</i> )
<b>Core i5</b> ( <i>Sexta Generación</i> )	<b>APU A10</b> ( <i>Séptima Generación</i> )
<b>Core i7</b> ( <i>Sexta Generación</i> )	<b>FX</b>

Y para servidores y maquinas de alta disponibilidad:

INTEL	AMD
<b>XEON</b>	<b>OPTERON</b>

Pero se hoy en día los móviles han superado con creces a las PC y laptops que ahora hablamos de otros proveedores para estos dispositivos:

ARM	INTEL	AMD	NVIDIA	Samsung	Qualcomm
Cortex-A73	Core M	Polaris	Tegra X1	Exynos 8	Snapdragon 820

## Dispositivos de Entrada/Salida

Para que el usuario pueda introducir la entrada, la computadora tiene varios **dispositivos de entrada** como parte de su **hardware**, como:

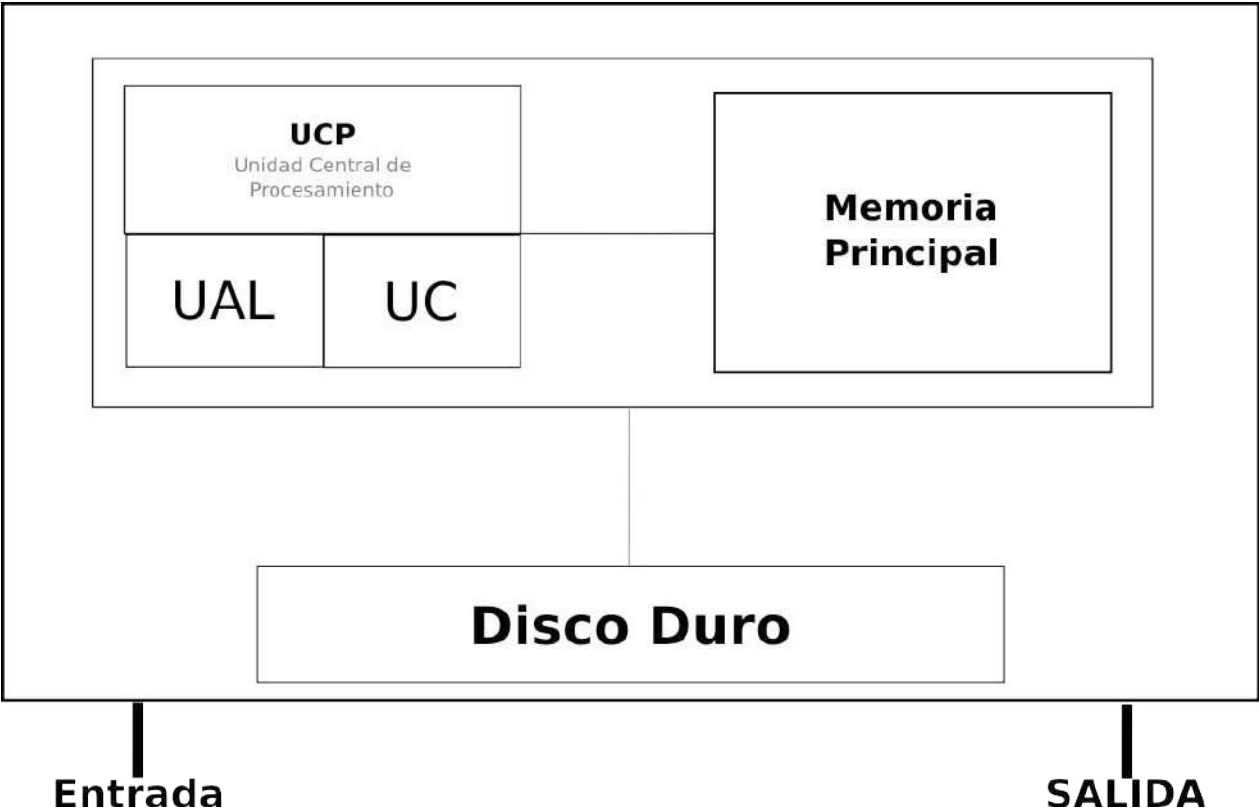
1. **Teclado**
2. **Ratón** (*Mouse*)
3. **Scanner**
4. **Unidad DVD**
5. **Unidad Blue Ray**
6. **Web Cam**
7. **Joystick**
8. **Lectores de Código de Barras**
9. **Micrófonos**

La salida al usuario se le proporciona mediante **dispositivos de salida**, tales como

1. **Monitor**
2. **Cañón**
3. **Impresora**
4. **Bocinas**

Los dispositivos de **entrada/Salida** permiten la comunicación entre la computadora y el usuario. Los Dispositivos de entrada como su nombre lo indica, sirven para introducir datos en la computadora para su procesamiento. Los datos se leen de los dispositivos de entrada y se almacenan en la memoria principal. Convirtiendo la información de entrada en impulsos eléctricos que son procesados por el sistema operativo.





# Introducción al lenguaje C

En esta parte nos adentraremos en el interesante mundo del lenguaje C, por un lado veremos algo de su historia, una parte de buenas practicas y lo mas importante es que ya empezaremos a ver código.

# Buenas practicas de programación.

Como lo mencione al principio, básicamente en este libro nos centraremos en aprender el lenguaje **C** para saber el **como** contribuir o empezar algún proyecto *open source*, aun hoy día existen tantas aplicaciones que usan el lenguaje **C** y es casi vital que sepamos programar en este lenguaje.

Pero muchos de nosotros no le tomamos la importancia necesaria a este tema hasta que intentamos contribuir en algún proyecto ya existente, obviamente al no llevar buenas practicas tendemos a ser regañados y en muchos casos humillados.

Algo que quiero que tomes a consideración es que en el mundo *Linux* la gran mayoría de usuarios que dicen usarlo son solo usuarios finales y cuando llegas a toparte con algún desarrollador son en muchos caso algo gruñones. Pero si quieres trabajar en este medio nunca tomes nada personal.

Y seguramente te vas a preguntar, por que razón varios desarrolladores son gruñones, bueno la respuesta a esto es bastante simple, lo pondré de la siguiente manera:

En muchos casos la mayoría de personas que aspiran a convertirse en desarrolladores suelen pedirles ayuda preguntando cosas sin antes consultar la documentación o buscar si en algún otro lado alguien mas llevo a la respuesta, básicamente les gusta lo fácil y los desarrolladores están tan ocupados tratando de solucionar un problema que simplemente les da flojera contestar o te dicen que leas el manual pero con mas palabras antisonantes.

Lo primero que tienes que entender es que a los desarrolladores les gustan los problemas realmente complejos y las buenas preguntas que les hagan pensar en ellos. De no ser así no estarían en donde están. Si les proporcionas preguntas interesante te lo agradecerán mucho inclusive podrían empezar a respetarte; ya que las buenas preguntas suponen un estimulo y un regalo. Las buenas preguntas ayudan al desarrollador a mejorar la comprensión, y a menudo revelan problemas que podrían no haber percibido o en los que de otra manera no habrían reparado. En el mundo del desarrollador las buenas preguntas podrían entenderse como un sincero cumplido.

Por otro lado como se relaciona esto con las buenas practicas en la programación, llegado a este momento ahora supondremos que intentas realizar una parche de tu programa favorito, pero tu nunca pusiste atención ni leíste su estándar de codificación y en donde aprendiste a programar nunca usaste estas buenas practicas. Al intentar subir tu cambio los desarrolladores verán que no cumple el estándar y aun que haga lo que se supone que debe hacer, no lo incluirán dentro del código principal.

No te quiero engañar este capítulo no te hará mejor programador solo intentara mostrarte algunos conocimientos de los principios, patrones y heurísticas propias de haber sufrido en carne propia la necesidad de usar buenas practicas. También no sera nada fácil si es que ya tienes un estilo de programación propio, difícil pero no imposible. Soy de la idea de que las cosas que experimenta uno mismo son las que mas nos permiten aprender por lo que en esta parte nos centraremos en ver el **como** de una manera practica.

Pero todo esto requiere algo mas que conocer principios y patrones, deberemos practicarlo y fallar y volver a intentarlo, incluso debes ver como otros practican y fallan recuperándose al paso, o ver el precio que pagan por tomar decisiones equivocadas.

Si eres un programador con algunos años de experiencia, probablemente ya sufriste los desastres cometidos por otros en el código. En un periodo de algunos años los equipos que avanzaban a pasos agigantados pueden acabar a pasos de tortuga. Cada cambio en el código afecta a dos o tres partes del mismo. Ningún cambio sera trivial. Y para ampliar o modificar el código sera necesario comprender todos los detalles, efectos y consecuencias, para de ese modo poder añadir nuevos detalles, efectos y consecuencias. Con el tiempo, el desastre aumenta de tal modo que no se puede remediar.

Al aumentar este desastre, la productividad del equipo disminuye y acaba por desaparecer. Al reducirse la productividad, el encargado hace lo único que puede; ampliar la plantilla del proyecto con la esperanza de aumentar la productividad. Pero esa nueva plantilla no conoce el diseño del sistema. No conoce la diferencia entre un cambio adecuado al objetivo de diseño y otro que lo destruya. Por tanto todos se encuentran sometidos a una gran presión para aumentar la productividad. Por ello, cometen mas errores, aumentando el desastre y la productividad se acerca a cero cada vez mas y mas.

En ultima instancia, el equipo se rebela. Informan al encargado que no pueden seguir trabajando con ese código. Exigen un cambio de diseño. La dirección no requiere invertir en un cambio de diseño del proyecto pero no pueden ignorar el bajo nivel de productividad. Acaban por ceder a las exigencias de los programadores y autoriza el gran cambio de diseño. Se selecciona un nuevo equipo. Todos requieren formar parte del nuevo equipo por ser lienzo en blanco. Pueden empezar de cero y crear algo realmente bello, pero solo los mejores serán elegidos para el nuevo equipo. Los demás deben continuar con el mantenimiento del sistema actual.

Ahora los 2 equipos compiten. El nuevo debe crear un sistema que haga lo que el antiguo no puede. Además, debe asumir los cambios que continuamente se aplican al sistema antiguo. El encargado no sustituirá el sistema antiguo hasta que el nuevo sea capaz de hacer todo lo que hace el antiguo.

Esta competición puede durar mucho tiempo. Y cuando acaba, los miembros originales del equipo se encuentran trabajando en otras empresas y los miembros actuales exigen un cambio de diseño del nuevo sistema por que es un desastre.

Si ya sufriste alguna fase de esta breve historia, ya sabrás que dedicar tiempo a que el código sea correcto no solo es rentable, es una cuestión de supervivencia profesional.

Imaginemos que creemos que las buenas practicas no son un obstáculo significativo. Imaginemos también que aceptamos que la única forma de avanzar es mantener esas buenas practicas. Entonces te preguntarás como puedo crear buen código. No tiene sentido intentar crearlo si no sabemos lo que es. La mala noticia es que crear buen código es como pintar un cuadro. Muchos sabemos si un cuadro se ha pintado bien o no, pero reconocer la calidad de una obra no significa que podamos pintar. Por ellos, reconocer buen código no significa que sepamos como crearlo.

Un programador sin este sentido puede reconocer el desastre cometido en un modulo pero no saber como solucionarlo. Y un programador con este sentido vera las posibles opciones y elegirá la variante optima para definir una secuencia de cambios.

**En definitiva, un programador que tengas buenas practicas es un artista que puede transformar un lienzo en blanco en un sistema de código elegante.**

Y este es uno de los motivos por el que los proyectos *open source* han logrado sobrevivir tantos años, inclusive al pasar a través de encargados diferentes. *Linux* y muchos proyectos son muestra de ello.

## Nombres consistentes.

Por lo general al momento de codificar algo, asignamos nombre a archivos, carpetas y archivos. Usamos nombres constantemente. Por ello, debemos hacerlo bien.

Es fácil afirmar que los nombres deben revelar nuestras intenciones. Lo que quiero recalcar es la importancia de hacerlo. Elegir nombres correctos lleva tiempo pero también ahorra trabajo. Por ello, prestar atención a los nombres es vital.

El nombre de una variable, función, o clase debe responder una serie de cuestiones básicas. Debe indicar por que existen, que hace y como se usa. Si un nombre requiere un comentario, significa que no revela su contenido.

Incorrecto:

```
int d; //indica el mi edad
```

Correcto:

```
int edad;
```

Como viste arriba el nombre de la variable "d" no indica nada. No evoca la sensación de un valor que pueda describir mi edad.

Otro mejor ejemplo podría ser el siguiente:

```
int edadDias;  
int edadHoras;  
int edadSegundos;  
int edadAños;
```

Debes evitar dejar pistas falsas el significado del código, cuyo significado se aleje del que pretendemos. Por ejemplo:

```
int aix;  
int hp;  
int oracle;  
int solaris;
```

Que son nombres de variables pobres ya que son los nombres de otros sistemas operativos.

Gran parte de nuestro cerebro se dedica al concepto palabra. Y, por definición, las palabras son pronunciables. Sería una pena malgastar esa parte de nuestro cerebro dedicada al lenguaje hablado. Por tanto es importante también usar nombres que sean pronunciables. Esta es una actividad importante, ya que la programación es una actividad social.

Incorrecto:

```
int vsdincper; //Variable que guarda solo el día del nacimiento de la persona
```

Correcto:

```
int diaNacimPersona;
```

Como puedes ver quizá sea un poco grande pero esa variable al leerla es fácil y te dice que valor guarda.

Evita usar la misma palabra para 2 fines distintos. Si aplicas la regla de una palabra por conceptos, acabaras con muchos elementos que por ejemplo tengan una palabra *agregar*. Mientras las listas de parámetros y los valores devueltos de las distintas funciones *agregar* sean semánticamente equivalentes, no hay problema. Sin embargo podrías usar la palabra *agregar* por motivos de coherencia, aunque no sea en el mismo sentido. Imagina que hay varias funciones en las que *agregar* puede crear un nuevo valor sumando variables o concatenando 2 cadenas. Imagina ahora creamos una nueva función que añade una parámetro a una colección. Parece coherente ya que hay muchas otras funciones *agregar*, pero en este caso hay una diferencia semántica, de modo que debemos usar un nombre como *insertar* o *añadir*. Ten en consideración que como desarrolladores y autores de algún programa, nuestro objetivo es facilitar la comprensión del código.

Recuerda que las personas que quizás vayan a leer tu código serán programadores. Por ello, es recomendable usar términos matemáticos, algoritmos, nombres de patrones y demás.

Algunos nombres tienen significado por si mismos, pero la mayoría no. Por ello, debemos incluirlos con un contexto, variables, funciones y espacios de nombres con nombres adecuados. Cuando todo lo demás falle, pueden usarse prefijos como ultimo recurso.

Imaginemos que tenemos la variable `primerNombre` , `ultimoNombre` , `calle` , `numeroCasa` , `ciudad` , `estado` , `codigoPostal` . Si las combinamos, es evidente que forman una dirección. Pero si las variable `state` se usa de forma aislada en una función, ¿Sabríamos que es parte de una dirección? lógicamente podemos añadir contextos por medio de prefijos: `agregarPrimNombre` , `agregarUltimNombre` , `agregarEstado` . Al menos si los lectores del código comprenderán que estas variables forman parte de una estructura mayor.

## Manejo de las funciones

Desde los inicios de la programación, creábamos sistemas a partir de rutinas y subrutinas. Después de la época de **Fortran** y **PL/1**, se creaban sistemas con programas, subprogramas y funciones. En la actualidad, solo las funciones han sobrevivido. Son la primera linea organizativa en cualquier programa en varios proyectos *Open Source*.

La primera regla de las funciones es que deben ser de tamaño reducido.

La segunda es que deben ser todavía mas reducidas.

Quizás no sea una afirmación que pueda justificar. Del mismo modo no puedo mostrar referencias a estudios que demuestren que las funciones muy reducidas sean mejores. Lo que si puedo afirmar es que durante casi 7 años he creado funciones de diferentes

tamaños. Desde monstruosas funciones de 300 a 400 líneas a funciones de 30 a 100 líneas. Esta experiencia me ha demostrado, mediante ensayo y error, que las funciones deben ser muy reducidas.

En la década de los 80's se decía que una función no debía superar el tamaño de una pantalla. Que por aquellos tiempos tenían 24 líneas por 80 columnas. En la actualidad, con una fuente mínima y un monitor de gran resolución, se puede ver 180 caracteres por líneas y 120 líneas o mas en monitores 4k.

Lo que nos deja con la siguiente interrogante:

¿Qué tamaño mínimo debe tener una función?

Quizás para responder esta interrogante debas considerar una de las premisas de **UNIX**, pero modificada un poco para entender este contexto:

Que cada función solo debe hacer una cosa. Debe hacerlo bien y debe ser lo único que haga.

De igual manera que las variables las funciones necesitan nombres consistentes. Cuanto mas reducida y concreta es una función, mas sencillo sera elegir un nombre descriptivo. No temas a los nombres extensos. En muchos casos un nombre extenso es preferible sobre uno breve y enigmático. Lo mejor seria usar una convención de nombres que permita leer varias palabras en los nombres de la funciones y usa esas palabras para asignar a la función un nombre que describa su cometido.

Correcto:

```
int sumaEnteros()
```

Incorrecto:

```
int suEN()
```

La elección de nombres descriptivos clarifica el diseño de las funciones y le permite mejorarlos. Se coherente con los nombres. Eso quiere decir que uses las mismas frases, sustantivos y verbos en los nombres de función que elijas para las funciones.

## Argumento de funciones.

El numero ideal de argumentos para una función es lo menor que se pueda. Ya que los argumentos son todavía mas complicados desde un punto de vista de pruebas. Imagina la dificultad de crear todos los casos de prueba para garantizar el funcionamiento de las



distintas combinaciones de argumentos. Si no hay argumentos, todo sera mas sencillo. Si hay uno, no es demasiado difícil. Con 2 o 3 argumentos el problema es mas complejo. Con mas de 3 argumentos, probar cada combinación de valores adecuados es todo un reto. Los argumentos de salida son mas difíciles de entender que los de entrada. Ya que al leer una función, estamos acostumbrados al concepto de información añadida a la función a través de argumentos y extraída a través de un valor de retorno. No esperamos que la información se devuelva a través de los argumentos. Por ellos, los argumentos de salida suelen obligarnos a realizar una comprobación doble.

Hay 2 motivos principales para pasar un solo argumentos a una función.

# Historia del lenguaje C

Desde el primer momento en que se necesitó algo menos complicado para poder desarrollar, los mismo desarrolladores empezaron a idear lenguajes que les facilitaran la vida a los que apenas empezaban de ahí que muchos lenguajes fueran creados con distintos propósitos.

*El lenguaje C ya tiene sus años. Su nacimiento está estrechamente vinculado al de sistema operativo **UNIX**. El investigador **Ken Thompson**, de **AT&T**, la compañía telefónica estadounidense, se propuso diseñar un nuevo sistema operativo a principio de los setenta. Disponía de un **PDP-7** en el que codificó una primera versión de **UNIX** en el lenguaje ensamblador. Pronto se impuso la conveniencia de desarrollar el sistema en un lenguaje de programación de alto nivel, pero la escasa memoria del PDP-7 (8K de 18bits) hizo que ideara el lenguaje de programación **B**, una versión reducida de un lenguaje ya existente: BCPL. El lenguaje **C** apareció como un **B** mejorado fruto de las demandas impuestas por el desarrollo de **UNIX**. Dennis Ritchie fue el encargado del diseño del lenguaje **C** y de la implementación de compilador sobre un PDP-11.*

Desde sus comienzos ha sufrido numerosos cambios a lo largo de su historia. La primera versión *estable* data de 1978 y se conoce como **K&R C**, es decir, **C de Kernighan y Ritchie**. La adopción de **UNIX** como sistema operativo de referencia en las universidades en los años 80 popularizó enormemente el lenguaje. No obstante, **C** era atractivo por sí mismo y parecía satisfacer una demanda real de los programadores de disponer de un lenguaje de alto nivel con ciertas características propias de los lenguajes de bajo nivel (*de ahí que a veces se diga que **C** es un lenguaje de nivel intermedio*).

La experiencia con lenguajes de programación diseñados con anterioridad como *Lisp* o *Pascal*, demuestran que cuando el uso de un lenguaje se extiende es muy probable que proliferen variedades dialécticas y extensiones para aplicaciones muy concretas, lo que dificultó enormemente el intercambio de programas entre diferentes grupo de programadores. Para evitar este problema se suele recurrir a la creación de un comité de expertos que define la versión oficial del lenguaje. El comité **ANSI X3J9** (*American National Standard Institute*), creado en 1983, considera la inclusión de aquellas extensiones y mejoras que juzgara de suficiente interés para la comunidad de programadores. El **14 de Diciembre de 1989** se acordó que era el **C estándar** y se publicó el documento con la especificación en la primavera de 1990. El estándar se divulgó con la segunda edición de **The C Programming Language**, de **Brian Kernighan** y **Dennis Ritchie**. Un comité de la international Standards Office (OSI) ratificó el documento del comité de ANSI en 1992, convirtiéndolo así en un estándar internacional. Durante mucho tiempo se conoció a esta

versión del lenguaje como **ANSI-C** para distinguirla así del **K&R C**. Ahora a la especificación de 1989 se le denomina **C89** para distinguirla de la revisión que se publicó en 1999, llamada **C99** que fue la que se usó mucho por defecto para el compilador **GCC** hasta su versión 4.9, para 2011 se realizó otra revisión del estándar y se produjo **C11** y es la versión estándar de **C** que estudiaremos.

**C** ha tenido un gran impacto en el diseño de otros lenguajes. Ha sido, por ejemplo, la base para definir la sintaxis y ciertos aspectos de la semántica de lenguajes tan populares como **java** o **C++**.

# Introducción