



MODULO 1

Introducción a la programación

Breve descripción

En este documento se desarrollan el concepto de qué es la programación, qué es un compilador y nuestros primeros programas en C.

Versión 1.1

Martin Jerman

Martin.jerman@inspt.utn.edu.ar



Introducción

La computadora es una herramienta poderosa. La información (datos de **entrada**) puede almacenarse en la memoria y manejarse a velocidades excepcionalmente altas (**proceso** de datos) para producir resultados (**salida** del programa). Podemos describirle a la computadora una tarea de manejo de datos presentándole una lista de instrucciones (llamada programa) que deben ser llevadas a cabo. Una vez que esta lista le ha sido proporcionada, ésta puede llevar a cabo (ejecutar) dichas instrucciones. El proceso de elaboración de dicha lista de instrucciones, o sea escribir un programa, se le llama **programación**. Escribir un programa de computadora es muy similar a describirle las reglas de un juego a gente que nunca lo ha jugado, para que las aplique. En ambos casos se requiere de un lenguaje de descripción intangible por todas las partes involucradas en la comunicación. Por ejemplo, las reglas del juego deben ser escritas en algún lenguaje y después se leen y se aplican. Los lenguajes utilizados para la comunicación entre el hombre y la computadora se llaman **lenguajes de programación**. Todas las instrucciones presentadas deben ser representadas y combinadas (para formar un programa) de acuerdo con las **reglas de sintaxis** (gramática) del lenguaje de programación. Sin embargo, hay una diferencia significativa entre el lenguaje de programación y un lenguaje como el español, el inglés o el ruso: las reglas de un lenguaje de programación son muy precisas y no se permiten excepciones o ambigüedades.

Los lenguajes de programación se pueden clasificar por niveles, según el grado de abstracción:

Lenguaje de Máquina	Lenguaje de Bajo Nivel	Lenguaje de Alto Nivel
Directamente interpretable por un circuito microprogramable, como el microprocesador de una computadora o el microcontrolador de un autómata (un PLC). Los circuitos microprogramables son sistemas digitales, lo que significa que trabajan con dos únicos niveles de tensión. Dichos niveles, por abstracción, se simbolizan con el cero, 0, y el uno, 1, por eso el lenguaje de máquina sólo utiliza dichos signos. Esto permite el empleo de las teorías del álgebra booleana y del sistema binario en el diseño de este tipo de circuitos y en su programación. El lenguaje de programación de primera generación (por sus siglas en inglés, 1GL), es el lenguaje de código máquina. Es el único lenguaje que un microprocesador entiende de forma nativa.	Es el que proporciona poca o ninguna abstracción del microprocesador. Es fácilmente trasladado a lenguaje de máquina. La palabra "bajo" no implica que el lenguaje sea inferior a un lenguaje de alto nivel; se refiere a la reducida abstracción entre el lenguaje y el hardware. El lenguaje de programación de segunda generación (por sus siglas en inglés, 2GL), es el lenguaje ensamblador. Se considera de segunda generación porque, aunque no es lenguaje nativo del microprocesador, un programador de lenguaje ensamblador debe conocer la arquitectura del microprocesador (como por ejemplo las particularidades de sus registros o su conjunto de instrucciones).	Se caracterizan por expresar los algoritmos de una manera adecuada a la capacidad cognitiva humana, en lugar de a la capacidad ejecutora de las máquinas. En los primeros lenguajes de alto nivel la limitación era que se orientaban a un área específica y sus instrucciones requerían de una sintaxis predefinida. Su función principal radica en que a partir de su desarrollo, existe la posibilidad de que se pueda utilizar el mismo programa en distintas máquinas, es decir que es independiente de un hardware determinado. La única condición es que la PC tenga un programa conocido como traductor o compilador, que lo traduce al lenguaje específico de cada máquina. Existe gran diversidad de ellos (C, PASCAL, BASIC, C++, VISUAL BASIC, COBOL, etc).



Compilación vs Interpretación

Entre los lenguajes de alto nivel existe una clasificación según cómo las instrucciones son ejecutadas. Dicha clasificación es:

- **Lenguajes compilados:** estos lenguajes requieren de un proceso llamado compilación, que consiste en que las instrucciones descritas en el código fuente se traduzcan al código de máquina para que puedan ser ejecutadas por una computadora. Un ejemplo de ello es el lenguaje C o C++.
- **Lenguajes interpretados:** estos lenguajes son traducidos a un código intermedio que es ejecutado en una máquina virtual. Dicha máquina virtual debe estar instalada en la computadora donde se quiere ejecutar el programa. Este es el caso de Java.

En este curso utilizaremos el lenguaje C, un lenguaje que requiere de compilación para generar programas.

El proceso de compilación

Para empezar el proceso de programación, primero debemos utilizar un editor de texto o un entorno de desarrollo integrado (IDE) para escribir el programa. Una vez escrito el programa, se guarda el archivo generando así el archivo fuente o **código fuente**. A pesar de haber escrito el programa siguiendo las reglas del lenguaje, para la computadora sigue siendo una serie de caracteres, es decir que no es ejecutable. Para ello hay que compilarlo, que no es más que traducir esas instrucciones al código de máquina (una serie de unos y ceros entendibles por la computadora).

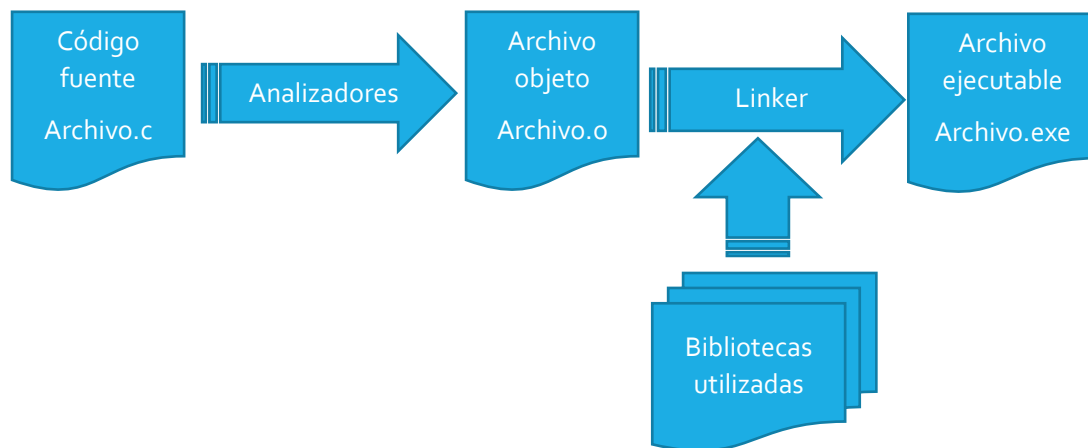
Al compilar un programa, el compilador pre-procesa el código fuente. Dicha etapa de pre-proceso consiste en correr 3 analizadores para validar que el programa escrito sea correcto:

- Analizador léxico
 - Agrupa los símbolos del programa fuente en unidades léxicas denominadas tokens.
 - Elimina los comentarios.
 - Elimina los espacios en blanco, retornos de carro, tabuladores, etc.
 - Agrega los identificadores a la tabla de símbolos.
 - Avisa de los errores léxicos que detecte.
- Analizador sintáctico
 - Crea un árbol sintáctico a partir de la secuencia de tokens creados en la fase de análisis léxico.
 - El árbol sintáctico sirve como base para el análisis semántico y la generación de código.
 - Avisa de los errores sintácticos que detecte.
- Analizador semántico
 - Comprueba que una frase sintácticamente correcta lo es también semánticamente.
 - Ejemplo asignación de valores de distintos tipos.
 - Ejemplo aplicación de operadores a tipos apropiados.
 - Avisa de los errores semánticos que detecte.

Una vez validado el código fuente, el compilador compila y genera un **código objeto**. Generalmente este archivo generado tiene la extensión ".o". Este código es la compilación del código fuente únicamente al cual todavía le falta el añadido de las bibliotecas que utiliza, por lo cual aún no es ejecutable.



Finalmente, el compilador hace uso de un **linker** o enlazador, que une el código objeto con las bibliotecas que éste llama y genera el definitivo **archivo ejecutable** (que en los sistemas Windows son los ".exe").



Lenguajes de programación

Los lenguajes de programación sirven para escribir programas que permitan la comunicación hombre-máquina y presentan las siguientes diferencias esenciales con el lenguaje natural:

Tienen un vocabulario y una sintaxis muy limitados, lo cual implica que los programas sólo pueden describir **algoritmos** y son inadecuados para describir otros tipos no algorítmicos de prosa.

1. Los objetos se declaran y las acciones se describen. Todo objeto referenciado en alguna parte de un programa debe haber sido definido previamente en el área reservada a las declaraciones.
2. El vocabulario de un lenguaje de programación contiene sólo aquellos tipos de acciones básicas que una computadora puede entender y realizar y no otras, por ejemplo, un lenguaje de programación soporta las cuatro operaciones fundamentales, comparación, acciones propias del procesamiento de textos, de gráficos y acciones de entrada y salida. Las acciones que una computadora no puede realizar son muchas y variadas, por ejemplo, crear pinturas al óleo es una acción que no se encuentra dentro del vocabulario de una computadora.
3. La sintaxis de un lenguaje de programación es muy rígida y no permite ninguna variación de estilo, por ejemplo, el cálculo del cociente entre dos números se expresa como n_1/n_2 y no existe forma alternativa.

¿Qué es un algoritmo?

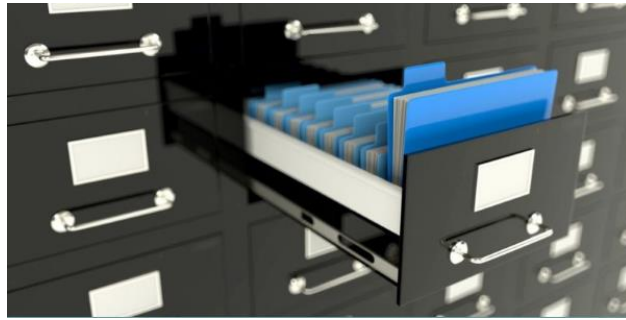
En matemáticas, lógica, ciencias de la computación y disciplinas relacionadas, un algoritmo es un conjunto de instrucciones o reglas definidas y no-ambiguas, ordenadas y finitas que permite solucionar un problema, realizar un cómputo, procesar datos y/o llevar a cabo otras tareas o actividades. Dados un estado inicial y una entrada, siguiendo los pasos sucesivos se llega a un estado final y se obtiene una solución.

Esta materia se centra en desarrollar la habilidad para comprender problemas, desentrañar lo que realmente es útil del problema (información) de lo que no lo es (dato) y crear un algoritmo que lo resuelva.



Datos vs Información

Un dato es una representación simbólica (numérica, alfabética, algorítmica, espacial, etc.) de un atributo o variable, que puede ser cuantitativa o cualitativa. Es decir que es un valor que por sí mismo no significa mucho. En nuestro caso, es un valor que recibe un programa por diferentes medios (teclado, archivos de entrada, etc.).



Ahora bien, cuando los datos se combinan en un contexto y son procesados de alguna manera (ya sea por un programa informático o algoritmo), se obtiene por resultado otros datos que además son información. Es decir que la información son datos que tienen un valor para quien procesa otros datos.

Paradigmas de programación

Un paradigma se refiere a una teoría o conjunto de teorías que sirve de modelo a seguir para resolver problemas o situaciones determinadas que se planteen. Es decir, es una forma de ver el problema para poder darle una solución. Dicho esto, en la historia de la programación hubo personas que concibieron ideas, conceptos y puntos de vista distintos para programar soluciones más simples.

Secuencial

Las acciones se ejecutan una tras otra, en un sentido estricto, esto es una acción posterior no se inicia hasta que se haya completado la anterior. Ejemplo: Basic, Cobol.

Estructurada

Es la escritura de programas de manera que el programa tiene un diseño modular: el código del programa está dividido en módulos o segmentos de código que son ejecutados conforme sea requerido; los módulos son diseñados de modo descendente: los problemas se dividen de tal manera que las partes complejas del programa sean implementadas (realizadas) por módulos independientes que a su vez pueden invocar a otros módulos. Ejemplo: Turbo Pascal.

Orientada a Objetos

Estilo de programación que utiliza objetos como bloque esencial de construcción. Los objetos son en realidad como los tipos abstractos de datos. Un TAD es un tipo definido por el programador junto con un conjunto de operaciones que se pueden realizar sobre ellos. Se denominan abstractos para diferenciarlos de los tipos de datos fundamentales o básicos.

El elemento básico de este paradigma no es la función (elemento básico de la programación estructurada), sino un ente denominado objeto. Un objeto es la representación de un concepto para un programa, y contiene toda la información necesaria para abstraer dicho concepto: los datos que describen su estado y las operaciones que pueden modificar dicho estado, y determinan las capacidades del objeto. Java incorpora el uso de la orientación a objetos como uno de los pilares básicos de su lenguaje.

Funcional

La programación funcional es un paradigma de programación declarativa basado en el uso de funciones matemáticas. Los lenguajes de programación funcional, especialmente los puramente



funcionales, han sido enfatizados en el ambiente académico y no tanto en el desarrollo comercial o industrial. Algunos ejemplos son el Scheme y el Haskell.

Lógico

La programación lógica es un tipo de paradigmas de programación dentro del paradigma de programación declarativa. La programación lógica gira en torno al concepto de predicado, o relación entre elementos. Un ejemplo de este paradigma es el Prolog.

Aprendiendo a programar

Por lo general, cuando una persona quiere explicar un procedimiento o proceso, suele describirlo como una serie de pasos. Es por eso por lo que para aprender a programar se utiliza el paradigma estructurado: porque es más fácil de visualizar cuando no se tiene una concepción previa de la programación.

En los lenguajes estructurados, así como en todos los lenguajes, existe una **sintaxis**. La sintaxis del lenguaje son reglas que deben seguirse en la escritura de cada parte de un programa. Siguiendo estas reglas el programar consiste en listar los pasos para resolver el problema en cuestión. Cada paso (o tarea) se lo llama **instrucción**. Las instrucciones se forman con los estatutos del lenguaje correspondiente y siguiendo las reglas de sintaxis que el mismo determine. De aquí se desprende que un programa es un conjunto de instrucciones.

El campeonato se gana "paso a paso", dijo Reinaldo "Mostaza" Merlo. Así se piensa al programar en C, paso a paso.

Buenas prácticas de programación

En todo lenguaje de programación existen reglas no obligatorias, pero si muy recomendables de escribir código. Estas reglas ayudan a que el código sea mantenible en el tiempo, legible por uno mismo y otros colegas. Para el caso de un lenguaje estructurado, dichas reglas o premisas son:

1. Modularidad a través del diseño top-down (diseño de lo general a lo particular)
2. Correcto indentado del código
3. Parametrizable (uso de variables y constantes)
4. Interfaz con el usuario amigable
 - a. entrada interactiva (letreros)
 - b. salida
5. Programación libre de errores
 - a. de entrada (manejo de errores)
 - b. lógicos (de programación)
6. Evitar uso de variables globales
7. Uso de nombres significativos de variables y módulos
8. Documentación que pueda leerse, usarse y modificarse por otros (del programa en comentarios en el propio código y/o documentos y también del uso general del mismo)
9. Manual de usuario

Ventajas de la modularidad:

- Construcción estratificada
- Depuración simple (por secciones)
- Fácil de leer



- Fácil de modificar
- Eliminación de código redundante
- Desarrollo independiente de módulos

Manejo de errores

Al momento de desarrollar programas es muy probable que se comenten errores. Estos errores pueden ser:

- Errores de sintaxis: son errores que aparecen en el código fuente por mala escritura de las instrucciones. Por ejemplo, una palabra reservada mal escrita.
- Errores en tiempo de compilación: son los errores que aparecen al momento de compilar el programa. Por lo general son errores de instrucciones inválidas (como una asignación de distintos tipos de datos o olvidarse un punto y coma).
- Errores de enlazado: puede que se quieran utilizar funciones que no estén disponibles por no agregar la biblioteca correcta. Por ejemplo, utilizar la función `printf` y no haber agregado el `#include <stdio.h>`.
- Errores de ejecución: aparecen cuando el programa se pudo compilar sin problemas, pero al ejecutarlo, éste falla o no se comporta como se espera. Por ejemplo, querer mostrar una variable real con formato de entero o intentar hacer una división por cero.
- Errores de diseño: son los errores más difíciles de resolver. Estos aparecen cuando se desarrolla un programa y no se contemplaron todos los casos de uso desde el principio, por lo que el error está en el planteo general del programa. Por lo general, estos errores o se parchean de alguna forma o fuerzan a reescribir todo el programa.

El lenguaje C

Historia

El lenguaje C nació en los Laboratorios Bell de AT&T y ha sido estrechamente asociado con el Sistema Operativo UNIX, ya que su desarrollo se realizó en este sistema y debido a que tanto UNIX como el propio compilador de C y la casi totalidad de los programas y herramientas de UNIX, fueron escritos en C. Su eficacia y claridad han hecho que el lenguaje Assembler apenas haya sido utilizado en UNIX. Está inspirado en el lenguaje B escrito por Ken Thompson en 1970 con intención de recodificar el UNIX, que en la fase de arranque está escrito en Assembler, en vistas a su transportabilidad a otras máquinas. B era un lenguaje evolucionado e independiente de la máquina, inspirado en el lenguaje BCPL concebido por Martin Richard en 1967. En 1972, Dennis Ritchie, toma el relevo y modifica el lenguaje B, creando el lenguaje C y reescribiendo el UNIX en dicho lenguaje. La novedad que proporcionó el lenguaje C sobre el B fue el diseño de tipos y estructuras de datos. Con la popularidad de las microcomputadoras muchas compañías comenzaron a implementar su propio C por lo cual surgieron discrepancias entre sí. Por esta razón ANSI (American National Standards Institute, por sus siglas en inglés), estableció un comité en 1983 para crear una definición no ambigua del lenguaje C e independiente de la máquina que pudiera utilizarse en todos los tipos de C. Algunos de los C existentes son: Quick C, C++, Turbo C, Turbo C ++, Borland C, Microsoft C, Visual C, C Builder.

Características principales

- Una de las particularidades de C es su riqueza de operadores, puede decirse que prácticamente dispone de un operador para cada una de las posibles operaciones en código máquina.



- C es un lenguaje de programación de nivel medio ya que combina los elementos del lenguaje de alto nivel con la funcionalidad del ensamblador.
- Es estructurado, es decir, el programa se divide en módulos (funciones) independientes entre sí.
- Actualmente, debido a sus características, puede ser utilizado para todo tipo de programas.
- Ha sido pensado para ser altamente transportable.

Inconvenientes

- Carece de instrucciones de entrada/salida, de instrucciones para manejo de cadenas de caracteres, con lo que este trabajo queda para la biblioteca de rutinas, con la consiguiente pérdida de transportabilidad.
- La excesiva libertad en la escritura de los programas puede llevar a errores en la programación que, por ser correctos sintácticamente no se detectan a simple vista.
- Por otra parte, las precedencias de los operadores convierten a veces las expresiones en pequeños rompecabezas.
- A pesar de todo, C ha demostrado ser un lenguaje extremadamente eficaz y expresivo.

Creación del programa

Los programas C y C++ se escriben con la ayuda de un editor de textos del mismo modo que cualquier texto corriente o utilizando un Entorno de Desarrollo Integrado (IDE). Los IDE proveen tanto las funciones de escritura de código, como también funcionalidades de compilación, depuración de código y resaltado de sintaxis, entre otras. Para programar en C recomendamos [Codeblocks](#); revisa el anexo de configuración del Codeblocks antes de empezar a programar.

Los archivos que contiene programas en C o C++ en forma de texto se conocen como archivos fuente y el texto del programa que contiene se conoce como programa fuente. Nosotros siempre escribiremos programas fuente y los guardaremos en archivos fuente.

El contenido del archivo deberá obedecer la sintaxis de C.

Los programas fuente no pueden ejecutarse. Son archivos de texto, pensados para que los comprendan los seres humanos, pero incomprensibles para las computadoras.

¡Entonces manos a la obra! Escribamos nuestro primer programa en C:

```
int main()
{
    int numero;
    numero = 2 + 2;
    return 0;
}
```

Estas simples líneas son un programa en C.

La primera línea `int main()` es el principio de la definición de una función. Todas las funciones C toman unos valores de entrada, llamados parámetros o argumentos, y devuelven un valor de retorno. La primera palabra: `int`, nos dice el tipo del valor de retorno de la función, en este caso un número entero. La función `main` siempre devuelve un entero. La segunda palabra es el nombre de la función, en general será el nombre que usaremos cuando queramos usar o llamar a la función. Por ahora quedémonos con esta idea dado que las funciones serán desarrolladas más adelante.



Si hay que destacar que todo programa en C tiene una función `main` dado que es la primera función por la que el programa empieza a ejecutar.

La segunda línea tiene solo un carácter, la apertura de llaves (`{`). En C, las llaves encierran el cuerpo de una función o agrupan instrucciones en un bloque. Por ende, si aparece una apertura de llaves, por consiguiente, al final de la función debe haber un cierre de llaves (`}`).

Luego en el cuerpo de la función principal se detallan las instrucciones del programa.

La tercera línea define una variable de tipo entero llamada `numero`. Esta es la forma en la que se declaran las variables en C. En algunos párrafos mas adelante se detallará como definir variables.

Luego, la siguiente línea suma `2+2` y guarda el resultado en la variable `numero`. Y finalmente la siguiente línea retorna el valor cero. Esto es así porque anteriormente la función `main` definimos que retornara un valor del tipo `int`.

Por último, la llave de cierre termina por cerrar la función `main` y también el programa.

Importante: todas las instrucciones en C terminan con un punto y coma `;`. Si llegase a faltar un punto y coma, el programa **no compila**.

Toda instrucción en C debe finalizar con un punto y coma ";", sino el programa no compila.

Cabe notar que las 3, 4 y 5 **están indentadas**, es decir que están tabuladas una posición hacia adentro. Esto es muy importante de tener en cuenta ya que ayuda a identificar rápidamente qué instrucciones corresponden a qué bloque de código. En este ejemplo simple no se aprecia su utilidad, pero a medida que los programas se hagan más largos nos será de gran ayuda.

Variables

La tercera línea define una variable de tipo entero llamada `numero`. Una variable es un espacio en memoria que el programador reserva para guardar un dato que recibe y que necesitará utilizar más tarde; es una porción de memoria que ningún otro programa podrá utilizar. Para el compilador cada variable es una dirección de memoria específica en donde se aloja el dato guardado. Dicho espacio de memoria se reserva cada vez que se ejecuta el programa. El espacio a reservar para cada variable depende de su tipo.

La forma en la que se declaran las variables en C es la siguiente:

```
<tipo de variable> <nombre de variable>;
```

Y los varios tipos de variables disponibles con los cuales trabajar son:

<code>char</code>	De 1 byte de tamaño, suele usarse para guardar caracteres
<code>int</code>	De 4 bytes de tamaño, suele usarse para guardar números
<code>float</code>	De 4 bytes de tamaño, para guardar números decimales
<code>double</code>	De 8 bytes de tamaño, para guardar números decimales
<code>short</code>	De 2 bytes de tamaño, suele usarse para números, aunque no tiene mucho uso
<code>long</code>	De 4 bytes de tamaño, suele usarse para números, aunque no tiene mucho uso

El valor almacenado puede cambiar en el transcurso de la ejecución del programa, pero siempre serán valores del tipo de dato al que pertenecen.



Cabe aclarar que todos los tipos de dato son **signados**, es decir que tienen signo (positivo o negativo). Por tanto, en el caso del `char` que es de 1 byte (es decir 8 bits) tenemos un bit para guardar el signo y 7 bits para guardar el dato. Con esta configuración tenemos espacio para guardar valores desde -128 hasta 127. ¿Qué pasa si quiero guardar el -129? Pues el valor se va del rango y se pierden bits, por lo que se guardara el valor 127. Esto mismo aplica a cualquier tipo de dato.

Tener muy en cuenta el rango de valores a guardar en una variable dado que éste depende del tipo de dato.

Que es un tipo de dato?

Las variables se definen por los tipos de datos que indicamos al crearlas.

Un tipo de dato define:

- El tamaño que dicho tipo va a ocupar en la memoria.
- el rango de valores que puede almacenar dicho tipo.
- La forma en que se almacenan en memoria los diferentes valores.
- Las operaciones que pueden realizarse con él.

Especificadores de variables

Los especificadores son palabras reservadas que sirven para explicitar alguna variante del tipo de dato o variable. Los especificadores son:

- Signed/unsigned: fuerza a la variable a tener o no tener signo. Por ejemplo, `signed char x`; fuerza a que `x` tenga signo, por lo que el intervalo de valores va de -128 a 127. De haber sido `unsigned`, el intervalo seria de 0 a 255.
- Long: aumenta el tamaño por defecto del tipo de dato. Solo aplica al tipo de dato `int` y `double`.
- Short: reduce el tamaño en bytes del tipo de dato. Solo aplica a `int` (lo convierte a 2 bytes).
- Const: fuerza a que, al asignarse un valor a la variable, este no se puede cambiar.
- Auto: especifica que la variable es local. No se utiliza porque toda variable que se declara es local automáticamente.
- Extern: sirve para especificar que la variable que se declara no es local, es decir que ya esta declarada en otro contexto/alcance y que se debe utilizar el valor de aquella (variable global). No es recomendable utilizar este tipo de variables.
- Static: este especificador actúa según la variable. En variables locales significa que la variable estará siempre en memoria como si fuera una variable global pero su alcance es el de una variable local. De este modo cuando se vuelve a ejecutar el bloque de código que incluye esta variable, la variable tendrá el valor que tuvo en su ultimo uso. En variables globales indica que dicha variable global es local al módulo del programa donde se declara, y, por tanto, no será conocida por ningún otro módulo del programa.
- Register: se aplica solo a variables locales de tipo `char` e `int`. Dicho especificador indica al compilador que, caso de ser posible, mantenga esa variable en un registro de la CPU y no cree por ello una variable en la memoria.

Cabe aclarar que algunos de estos especificadores se pueden combinar. Por ejemplo, `const unsigned char`;



Ámbito de las variables

El ámbito de una variable es la porción de código donde la variable está disponible (desde dónde se puede acceder a ella). Cuando una variable está disponible en una porción de código, diremos que es visible. Por ejemplo, en el siguiente código:

```
#include <stdio.h>
int main()
{
    int i=0;
    printf("Mostramos i\n");
    i+=50;
    printf("i=%d\n", i);
    return 0;
}
```

La variable `i` se encuentra visible dentro de `main`.

Estudiaremos dos ámbitos:

- Local: es el caso de la variable `i` del ejemplo anterior, donde dicha variable es visible dentro de `main`. Entonces se dice que “`i` es local a `main`”.
- Global: es el caso en que la variable está declarada fuera del bloque de código, pero es utilizada en dicho bloque. Lo veremos a continuación.

Variables globales

Las variables globales son aquellas que se definen **fuera** de la función `main`. Esto hace que su existencia y valor sea alcanzable desde cualquier parte del código, siempre y cuando la variable sea definida en ese bloque de código con el especificador `extern`. De no tener este especificador, la variable será una variable local. Estas variables pueden ser de cualquier tipo de dato soportado en C. Por ejemplo:

```
#include <stdio.h>
int unaVariable = 1;
int main()
{
    extern int unaVariable;
    return 0;
}
```

Como se ve en el ejemplo anterior, la variable es del tipo `int`, pero al estar definida fuera de todo bloque de código (en este caso, fuera de `main`) se dice que es global. Es importante saber que estas variables son visibles en todo el código. Para poder utilizarla, hay que definirla dentro de `main`, pero haciendo uso del especificador `extern`, que indica que dicha variable no hay que crearla sino utilizar la que ya está definida en otro ámbito más amplio.

El uso de variables globales está totalmente desaconsejado.

Variables constantes

El programador puede definir una variable constante la cual no podrá variar su valor una vez asignado. Por ejemplo, para crear una variable `float` correspondiente a la cotización dólar (la cual no puede modificarse durante el uso de la aplicación) escribimos:

```
const float cotizacion = 5.13;
```

Se puede usar `const` antes o después del tipo de dato, en el ejemplo utilizamos `float`. Generalmente se inicializa la variable al crearla dado que no podrá cambiarse de alguna otra forma. Pero si no se le asigna un valor, se le puede asignar uno más adelante; pero una vez asignado este no se podrá cambiar.



La directiva del preprocesador `#define` es similar a la utilización de constantes sólo que NO actúa como una variable. Al utilizar `#define` no se reserva espacio en memoria para un dato, sino que al momento de compilar se reemplaza el nombre definido por el valor indicado.

¡Hola mundo!

Ahora intentemos programar el ejemplo típico en el mundo de programación: el hola mundo. Veamos cómo es que se escribe:

```
#include <stdio.h>
int main()
{
    printf("hola mundo\n");
    return 0;
}
```

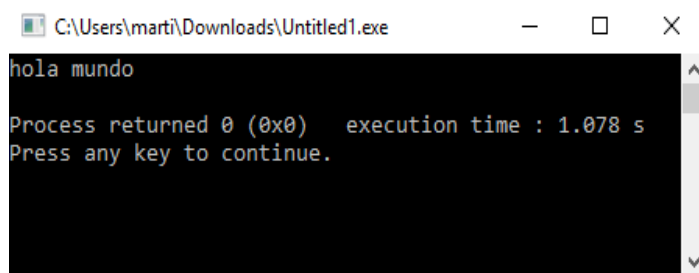
A diferencia del ejemplo anterior, en este caso la primera línea es una instrucción que inicia con el carácter `#`. Todas las sentencias que inician con este carácter son instrucciones para el **preprocesador**.

El preprocesador es el responsable de:

- Quitar los comentarios
- Interpretar las directivas del preprocesador las cuales inician con `#`. Por ejemplo:
 - `#include`: incluye el contenido del archivo nombrado. Estos son usualmente llamados archivos de cabecera (header), ejemplo `#include <math.h>` que es la biblioteca estándar de matemáticas o `#include <stdio.h>` que es la biblioteca estándar de Entrada/Salida.
 - `#define`: define un nombre simbólico o constante. Básicamente reemplaza un texto por el valor que se describe a continuación, ejemplo: `#define IVA 0,21`. En este caso se reemplaza la palabra IVA (literalmente, respetando las mayúsculas) con el número 0,21. Un uso avanzado de esta directiva es el de usar la frase como una función, por ejemplo `#define CUAD(x) (x*x)`. En este caso cuando tengamos por ejemplo una variable llamada `miVariable` y aparezca `CUAD(miVariable)`, se reemplazara por `(miVariable*miVariable)`.

Entrada y salida de datos

Luego en el cuerpo principal del programa tenemos dos instrucciones. La primera es `printf("hola mundo\n");` cuya función es mostrar el texto entre comillas por pantalla. Esta función no es parte de las funciones básicas del lenguaje C, por lo cual alguien la desarrolló y la incluyó en la biblioteca que ahora es estándar de C, la `stdio.h`. Es por eso por lo que siempre se incluye esta biblioteca (`stdio.h`) en todo programa en C. Existe otra biblioteca estándar llamada `stdlib.h` que también incluye algunas funciones útiles como las llamadas al sistema



Cuando se haga un copy/paste de un código al IDE para compilarlo, recuerda editar las comillas dobles dado que los procesadores de texto utilizan el símbolo de apertura y cierre; en cambio C utiliza la doble comilla estándar.



operativo con la función `SYSTEM()`. También es la suele incluir pero para el alcance de este curso no sera necesaria.

Siguiendo con la función `printf`, hasta ahora vimos como mostrar un texto por pantalla. Pero, cómo hacer para mostrar el valor de las variables? He aquí una instrucción de ejemplo:

```
int nroDeCaja = 4, cantidad = 8;

printf ("En la caja numero %d tenemos %d cantidad de manzanas\n", nroDeCaja,
cantidad);
```

En este ejemplo vemos cómo se utiliza la función `printf` para mostrar los valores de las variables `nroDeCaja` y `cantidad`. Observar que en la frase que se muestra hay dos `%d`. Este es el especificador de formato para mostrar la variable como un numero. Por cada `%d` que aparezca en la frase entre comillas dobles, debe haber una variable a mostrar y el orden de muestreo es **posicional**; en el primer `%d` se muestra la primer variable, en el segundo `%d` la segunda y así. De faltar variables o sobrar especificadores de formato, habrá un error de compilación.

Veamos a continuacion los formatos disponibles para mostrar variables:

Formato	Aplicación
<code>%c</code>	Muestra la variable según el código ASCII
<code>%d</code>	Muestra la variable en formato numérico entero
<code>%f</code>	Muestra la variable en formato decimal (flotante)
<code>%7.3f</code>	Muestra la variable en formato decimal especificando la cantidad de dígitos. En el ejemplo 7 es la cantidad total de dígitos incluido el punto a mostrar desde la derecha y 3 es la cantidad de decimales.
<code>%s</code>	Muestra los arreglos/vectores utilizados como cadenas (se verán mas adelante)

De forma analógica, existe una función para leer datos por teclado llamada `scanf()`. Esta función requiere que se especifique el formato de ingreso del dato y la variable en la que se guardara. Por ejemplo:

```
#include <stdio.h>
int main ()
{
    int miVariable = 0;
    printf("Ingrese un valor:\n");
    scanf("%d", &miVariable);
    printf("el valor ingresado fue %d\n", miVariable);
    return 0;
}
```

Observemos que en el formato especificado en la función, no hay un `\n`. Esto es porque de agregarlo, deberíamos dar dos veces el `Enter` para cargar el valor y además nos dará problemas para las siguientes lecturas.

Además, notemos el `&` delante del nombre de la variable (`&miVariable`). Por ahora tomemos como norma que antes de una variable siempre hay que agregar el `&` en el `scanf()`. Mas adelante veremos el por qué de esto y en qué casos no seria necesario.

Caracteres de escape

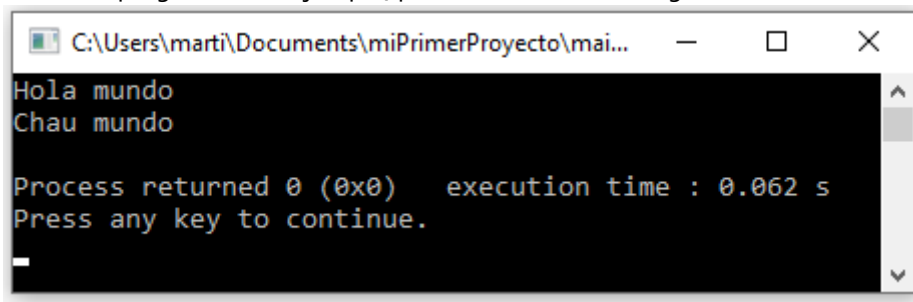
Siguiendo con el ejemplo anterior, verán que al ejecutar el programa se muestra la frase "hola mundo" pero sin ese `\n` del final. ¿Por qué es esto? Esto es porque en C es necesario explicitar cuando hay una tabulación o una nueva línea, etc. Estos caracteres empiezan con el carácter contra barra (`\`) seguidos por una letra. ¡Pero a pesar de que se escriben con dos caracteres, representan un único carácter!



Veamos a continuación la lista de caracteres de escape.

Código	Significado
\b	Retroceso
\f	Alimentación de hoja
\n	Nueva línea
\r	Retroceso de carro
\t	Tabulador horizontal
\"	Comilla doble
\'	Comilla simple
\0	Nulo
\\	Barra invertida
\v	Tabulador vertical
\a	Alerta
\o	Constante octal
\x	Constante hexadecimal

Es importante saber cómo funcionan los caracteres de escape, dado que ayudan a dar formato a la salida del programa. Por ejemplo, para obtener la salida siguiente:



Ambas soluciones siguientes son válidas:

```
#include <stdio.h>
#include <stdlib.h>
int main()
{
    printf("Hola mundo\n");
    printf("Chau mundo\n");
    return 0;
}
```

```
#include <stdio.h>
#include <stdlib.h>
int main()
{
    printf("Hola mundo\nChau mundo\n");
    return 0;
}
```

Operadores aritméticos

Al igual que todos los lenguajes de programación, C tiene los más usuales (suma +, resta -, multiplicación *, división / y modulo %).

Cabe aclarar el operador división es **división entera**. Pero el comportamiento puede variar según los operandos:

- Si la división se ejecuta entre dos valores o variables **enteros**, el resultado es un **entero** (si la división diera un valor decimal, el valor decimal se pierde).
- Si la división se ejecuta entre dos valores o variables **flotantes**, el resultado es un **flotante**.
- Si la división se ejecuta entre un valores o variable **entera** y un valor o variable **flotante**, el operando entero se promociona a flotante en esta instrucción únicamente para operarlo con el otro flotante y dar por resultado un **flotante**.



La promoción de tipos de dato ocurre porque las operaciones se deben realizar entre valores del mismo tipo de dato. La promoción sigue el orden los tipos de dato según su tamaño en bytes (ver sección *Variables*).

El operador asignación es un signo de igualación (=) y puede usarse al momento de declarar una variable para inicializarla o como una instrucción de asignación. Por ejemplo,

```
int a = 5;  
a = 4;
```

Además, C permite operadores de incremento solo para las variables enteras. Usualmente para incrementar una variable entera en 1 se escribiría

```
a=a+1;
```

Pero en C tenemos los operadores ++ y -. Estos operadores se pueden colocar antes o después de una variable entera, por ejemplo:

```
a++;  
--a;
```

Hay que aclarar que no es lo mismo a++; y ++a; El primer caso se conoce como pre incremento y en el segundo caso se conoce como post incremento. En ambos casos después de ejecutar la instrucción, la variable "a" quedara incrementada en 1. Pero en el pre-incremento la variable será incrementada en uno antes de ejecutar la instrucción. En cambio, en post incremento la variable es incrementada en 1 después de ejecutar la instrucción. Esto puede ser probado en un programa utilizando printf.

¿Y si queremos incrementar o decrementar en más de 1? Pues bien, para ello tenemos los operadores += y -=. Estos operadores incrementan la variable en la cantidad que se especifica a su derecha. Por ejemplo,

```
a += 4;  
miVariable -= 8;
```

Todos los operadores en C **devuelven un valor** como resultado de su ejecución y es un entero. Los operadores aritméticos devuelven el resultado de su ejecución. Podes probar esto haciendo printf de una expresión aritmética.

Operadores de comparación

El operador de igualdad es el doble signo igual (==) y es usado en las expresiones de las expresiones condicionales. Por el contrario, el operador de diferencia o distinto es la exclamación igual (!=).

Además, tenemos los demás operadores de comparación: <, <=, >, >=.

Como todo operador, estos los operadores de comparación devuelven un cero si la condición es falsa o un distinto de cero (por lo general es un 1 pero puede ser cualquier número) cuando la condición es verdadera. En C no existe el tipo de dato booleano.

En C no hay booleanos, por lo cual el valor entero cero se entiende como falso y lo distinto de cero es verdadero

Operadores lógicos

En C también tenemos los 3 operadores lógicos:

- OR (doble pipe ||)
- AND (el doble ampersand &&)
- NOT (la exclamación !).



Orden de precedencia

En el día a día todos estos operadores seguramente se utilizarán en expresiones largas donde se los combinará según sea necesario. ¿Cómo es que C decide cual aplicar primero? Para ello se C establece un orden de precedencia el cual se detalla en la siguiente tabla:

Operadores	Asociatividad
() [] ->	Izquierda a derecha
! - ++ -- + - * & (tipo) sizeof	Derecha a izquierda
* / %	Izquierda a derecha
+ -	Izquierda a derecha
<< >>	Izquierda a derecha
< <= > >=	Izquierda a derecha
== !=	Izquierda a derecha
&	Izquierda a derecha
^	Izquierda a derecha
	Izquierda a derecha
&&	Izquierda a derecha
	Izquierda a derecha
?:	Derecha a izquierda
= += -= *= /= %= &= ^= = <<= >>=	Derecha a izquierda
,	Izquierda a derecha

De acuerdo con lo anterior, la siguiente expresión: `a < 10 && 2 * b < c`

Es interpretada como: `(a < 10) && ((2 * b) < c)`

Notar que en la tabla aparecen operadores que no se describieron anteriormente. Para más información se recomienda buscar en el libro *El lenguaje de programación C* de Kernighan y Ritchie.

Comentarios

Siempre, en todo código, el programador necesita de agregar anotaciones o comentarios para ayudarlo a recordar qué es lo que hace el código o agregar información útil en el hipotético caso futuro de necesitar modificar algo. En C tenemos dos tipos de comentarios:

- De una sola línea: estos comentarios inician con una doble barra `//` y de ahí van hasta el final de la línea. Por ejemplo,

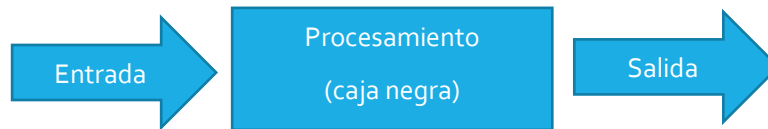
```
int a =1; // esto es un comentario de línea
```
- Por párrafos: estos comentarios encierran líneas. Para iniciar el comentario se utiliza el barra-asterisco `/*` y para cerrarlo el asterisco-barra `*/`. Usualmente se suele agregar un asterisco al principio de cada línea para enmarcar el comentario quedando, por ejemplo,

```
/* aquí empieza  *
 * El párrafo    *
 * De comentario */
```



Concepto de caja negra

En teoría de sistemas, una **caja negra** es un elemento que se estudia desde el punto de vista de las **entradas** que recibe y las **salidas** o respuestas que produce, sin tener en cuenta su funcionamiento interno. En otras palabras, de una caja negra **nos interesará su forma de interactuar con el medio** que le rodea (en ocasiones, otros elementos que también podrían ser cajas negras) entendiendo qué es lo que hace, pero sin dar importancia a cómo lo hace.



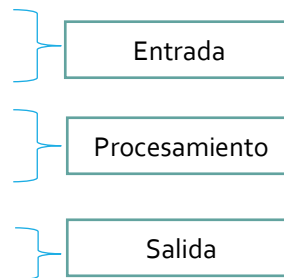
Por tanto, de una caja negra deben estar muy bien definidas sus entradas y salidas, es decir, su interfaz; en cambio, no se precisa definir ni conocer los detalles internos de su funcionamiento.

Esta idea es útil para plantear la idea del algoritmo que luego llevaremos al programa. Veamos el siguiente ejemplo:

```
#include <stdio.h>
int main ()
{
    int miVariable = 0;
    printf("Ingrese un valor: ");
    scanf("%d", & miVariable);

    miVariable++;
    miVariable--;
    miVariable *= miVariable;

    printf("el cuadrado es %d\n", miVariable);
    return 0;
}
```



La gran mayoría de los programas respetan esta estructura básica. De hecho, los programas que desarrollaremos en la práctica tendrán esta forma. Además, este concepto será muy útil para más adelante, cuando veamos funciones y procedimientos.