1. Computadores y programación

Informática y computadora (RAE)

Informática (Ciencia de la computación)

Conjunto de conocimientos científicos y técnicos que hacen posible el tratamiento automático de la información por medio de ordenadores.

Computadora

Máquina electrónica, analógica o digital, dotada de una memoria de gran capacidad y de métodos de tratamiento de la información, capaz de resolver problemas matemáticos y lógicos mediante la ejecución de programas informáticos.

Hardware y Software (RAE)

Hardware

Componentes que integran la parte física de una computadora.

Software

Conjunto de programas, instrucciones y reglas informáticas para ejecutar ciertas tareas en una computadora.

El Sistema Operativo

- Es el software básico para manejar el hardware y proporcionar un conjunto de servicios genérico al resto de programas, las aplicaciones.

- Es la primera capa de software que tiene una computadora, que establece un puente entre el hardware y el resto del software (aplicaciones)... pero en definitiva es un programa

- Sistemas operativos más usuales: Linux, Windows (en sus distintas versiones), Mac OS, Solaris, Unix, Android...

- En FP nos interesa aprender a diseñar aplicaciones. No vemos los sistemas operativos a fondo.

Programas, algoritmos, cómputos

Algoritmo

Descripción precisa y ordenada de una secuencia (finita) de operaciones que permite hallar la solución de un problema.

Conocemos algún algoritmo? multiplicación, división, mcd...
Al ser una secuencia finita de instrucciones, siempre terminará... es cierto? NO

Programa

Codificación de un algoritmo en un lenguaje de programación concreto (C#, C++, Java, ...), mediante una secuencia de instrucciones que "entiende" la computadora.

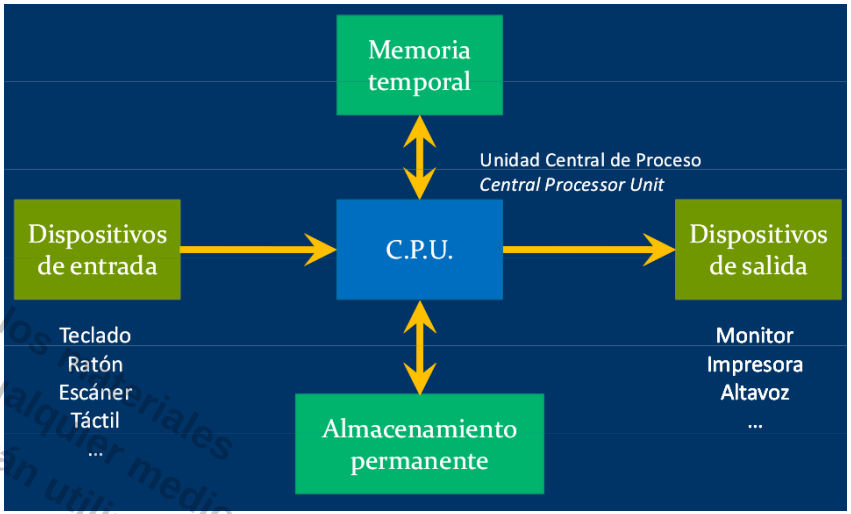
Son distintos programa y algoritmo? Conocemos algún programa?

Cómputo

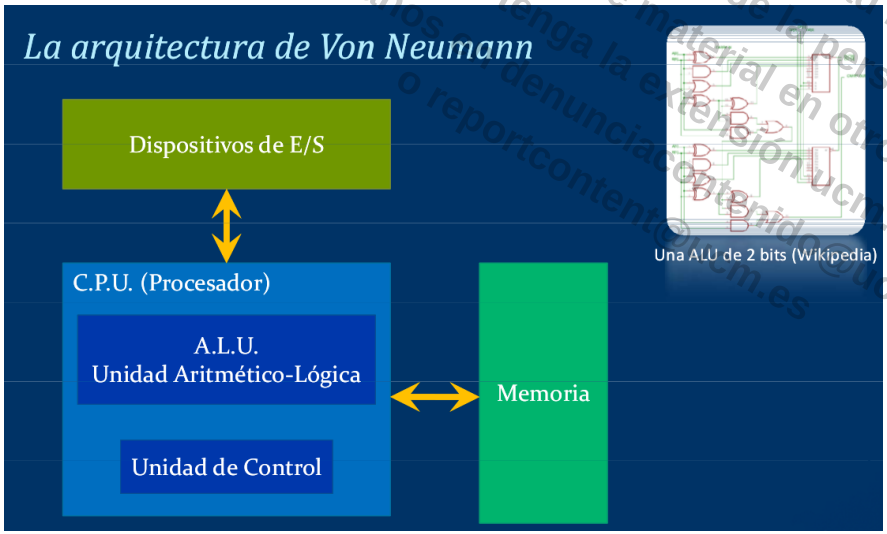
Ejecución de un programa en un ordenador.

Cuantos cómputos puede hacer un programa?

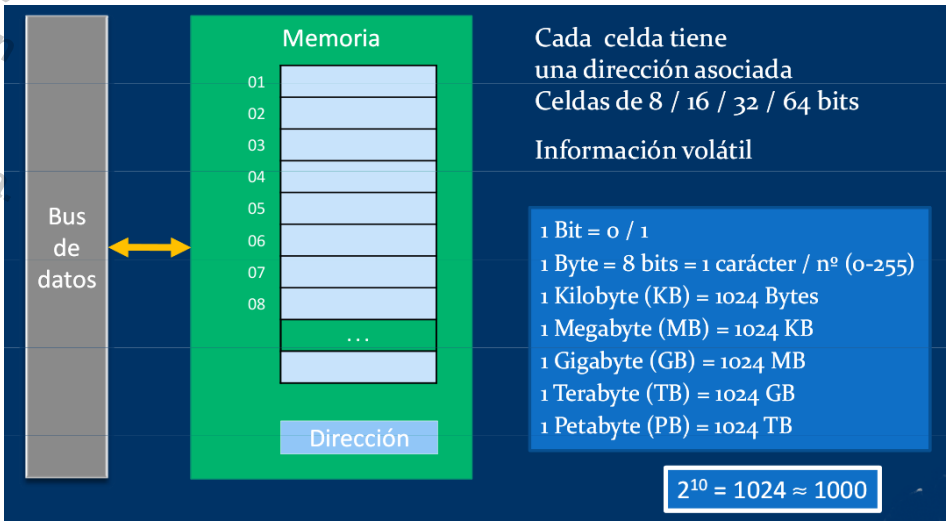
Arquitectura de una computadora



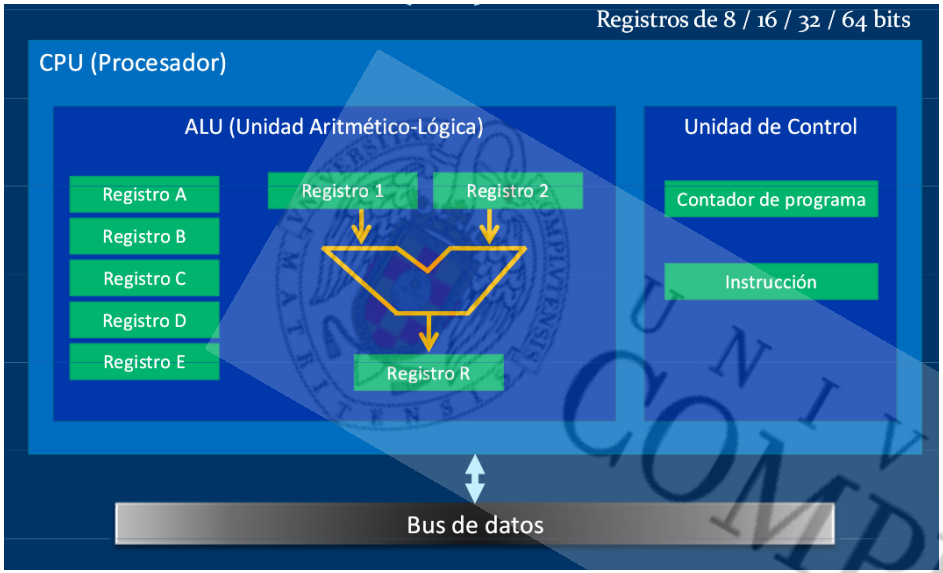
Arquitectura elemental



Memoria Principal



Unidad Central de Proceso (CPU)



Programas a bajo nivel: el lenguaje *máquina*

• La CPU trabaja en sistema binario, con ceros y unos (bits, unidad mínima de información/dato)
Byte: grupo de 8 bits (unidad básica de acceso a memoria)

En el lenguaje máquina TODO se representa en binario: instrucciones del programa, datos, direcciones de memoria, etc.

- Es habitual utilizar representación en sistema hexadecimal
  Así, el byte 01011011 se representa como:

$$\underbrace{0101}_{5}\underbrace{1011}_{B}$$

01011011 binario $\equiv 5B$ hexadecimal

Lenguaje máquina

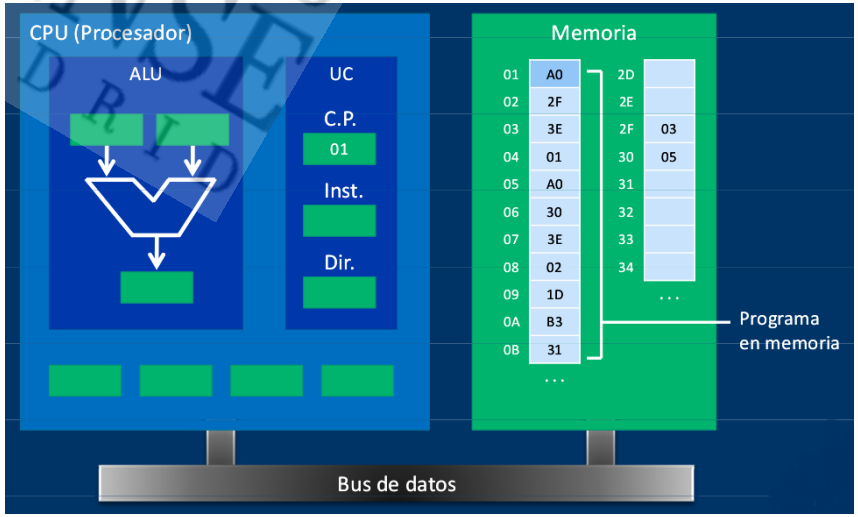
Un ejemplo de programa en lenguaje máquina para sumar dos números

Instrucción	Significado
A0 2F	Acceder a la posición de memoria 2F
3E 01	Copiar el dato en el registro 1 de la ALU
A0 30	Acceder a la posición de memoria 30
3E 02	Copiar el dato en el registro 2 de la ALU
1D	Sumar (registros 1 y 2)
B3 31	Guardar el resultado en la posición de memoria 31

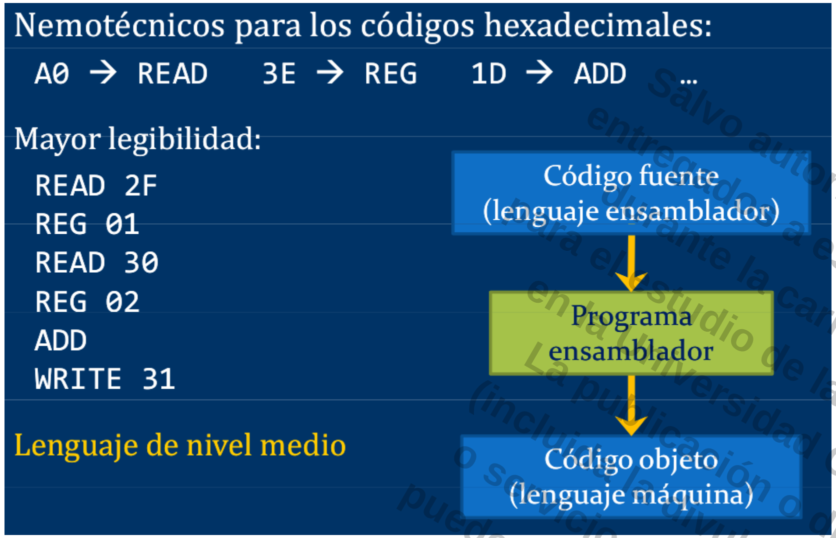
Este lenguaje es dependiente de la computadora concreta (de la CPU de la misma). Es distinto para un PC o un MAC... incluso es (ligeramente) distinto para cada CPU de esas arquitecturas.

Muy difícil programar directamente en lenguaje máquina.

Lenguaje máquina (II)



Lenguaje ensamblador. Un poco más fácil



Pero sigue siendo un lenguaje de bajo nivel, difícil de utilizar.

El siguiente paso: los lenguajes de alto nivel

- Más próximos al lenguaje natural y matemático

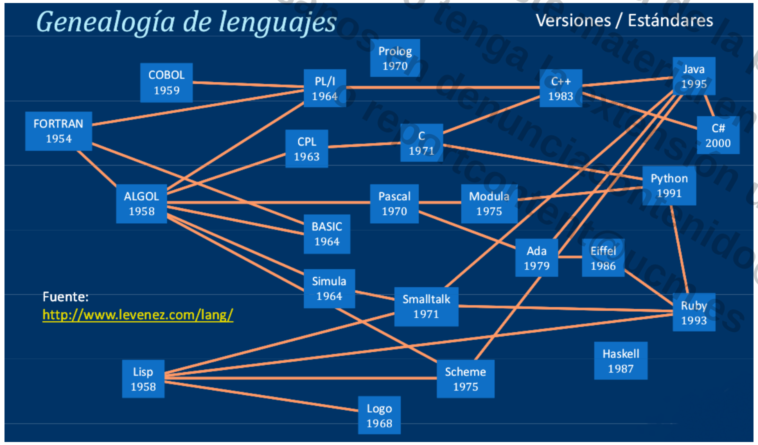
$$resultado = dato1 + dato2;$$

- Programas mucho más fáciles de escribir (y de leer)
- Después vino la programación estructurada, la abstracción procedimental, la estructuración de datos, la programación orientada a objetos...

  $\leadsto$ capas de abstracción para acercar el lenguaje a la persona y facilitar el diseño de programas.

- Infinidad de lenguajes: C++, Java, Python, Fortran, Prolog, Haskell, Pascal, Cobol, Lisp, Smalltalk, C#,...

Muchos lenguajes de programación



En `https://es.wikipedia.org/wiki/Anexo:Lenguajes_de_programacion` hay una lista muy incompleta con más de 600 lenguajes de programación.

¿Por qué C#?

- Lenguaje moderno (2000)
- Influido por lenguajes consolidados Java, C++, Eiffel, Modula-3, Pascal... incorporando lo mejor de ellos.
- Elecciones de diseño muy acertadas, diseñado por programadores experimentados como Anders Hejlsberg

  es elegante, simple, con tipos seguros...

- Completamente integrado en la plataforma .NET de MS Windows, pero multiplataforma
- Descriptores (Wikipedia) Multiparadigma: estructurado, imperativo, orientado a objetos, dirigido por eventos, funcional, genérico, reflexivo

Por qué se llama C#? # (sostenido musical, un semitono por arriba: superior a C). Es *otra* evolución de *C*.

¿Por qué C# en el grado de videojuegos?

- Porque se integra perfectamente con el motor Unity
- Es un lenguaje de propósito general... no sirve solo para videojuegos, sino para cualquier tipo de aplicación.
- Es un lenguaje con futuro y es perfecto para el currículum de un alumno de este grado.
- Es un buen lenguaje de iniciación a la programación... pero además es un lenguaje profesional!

Por otro lado:

- El objetivo prioritario del curso NO es aprender C#... es aprender programación estructurada: C# es muy razonable.
- Un programador debe conocer distintos (y variados) lenguajes de programación y aprender constantemente otros nuevos.
- ¿Cuál es el mejor lenguaje de programación?
  Cada programador tiene su(s) favorito(s). Lo ideal es dominar muchos y utilizar uno u otro dependiendo de la aplicación.

Sintaxis y semántica de los lenguajes

Un lenguaje (formal) queda definido (formalmente) por dos aspectos esenciales:

- Sintaxis: reglas que determinan las construcciones válidas del lenguaje. ¿Está bien escrito?
- Semántica: significado que se atribuye a las construcciones válidas del lenguaje. ¿Qué hace?

Un lenguaje de programación es un lenguaje formal.

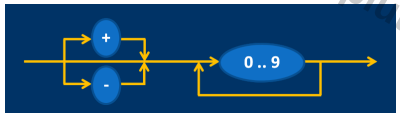
- La sintaxis determina lo que es un *programa* válido.
- La semántica determina lo que hace un programa válido (el resultado que produce su ejecución).

En lo sucesivo nos centramos en lenguajes de programación.

Sintaxis de los lenguajes de programación

Se define mediante Diagramas de flujo o Gramáticas

• Diagramas de flujo: representación gráfica de la forma de construcción de elementos del lenguaje. Por ejemplo, para los números enteros:



De acuerdo con esto, ¿están bien formados los siguientes números?

$$23, -159, +5, 1 - 34, 3{,}14, 002$$

Sintaxis de los lenguajes de programación. Gramáticas

La forma más habitual de definir la sintaxis es con

• Notación BNF (Bakus-Naur Form) y gramáticas libres de contexto: más formal, más potente, más precisa.
Por ejemplo, para los enteros:

entero	→	signoOp secDigitos
signoOp	→	*+* | *-* | ϵ
secDigitos	→	digito | digito secDigitos
digito	→	*0* | *1* | *2* | *3* | *4* | *5* | *6* | *7* | *8* | *9*

- Cada línea es una *producción gramatical*
- Los símbolos no terminales son los que tienen una producción asociada o más (los que aparecen a la izquierda de →)
- Los símbolos terminales son el resto.
- | significa *alternativa* (ó)
- ϵ significa *secuencia vacía*

Gramáticas. Reconocimiento, generación.

entero	→	signoOp secDigitos
signoOp	→	*+* | *-* | ϵ
secDigitos	→	digito | digito secDigitos
digito	→	*0* | *1* | *2* | *3* | *4* | *5* | *6* | *7* | *8* | *9*

De acuerdo con esto, ¿están bien formados los números 23, -159, +5, 1-34, 3.14, 002?

Por ejemplo, para el 23:

entero	→	signoOp secDigitos
	→	ϵ secDigitos $\equiv$ secDigitos
	→	digito secDigitos
	→	2 secDigitos
	→	2 digito
	→	2 3

Desarrollo de programas

Dos ideas previas sencillas:

- El objetivo de un programa es resolver un problema.
- El ordenador *no piensa*, solo ejecuta un programa dado.

Un programa tiene que estar bien escrito de acuerdo con la sintaxis del lenguaje para que pueda ser ejecutado.

Pero además debe ser correcto

¿Qué significa que un programa sea correcto?

Desarrollo de programas

Programas correctos (I)

Un programa es correcto si hace exactamente la tarea para la que ha sido diseñado.

Formalmente: es correcto si se comporta exactamente según una especificación dada. La especificación define o describe lo que debe hacer hacer el programa.

A su vez la especificación puede ser:

- Formal (utilizando lenguajes formales de especificación, que utilizan la lógica matemática). Por ejemplo, con el modelo de precondición y postcondición y la lógica de Hoare.
- Menos formal, utilizando lenguaje natural.

En cualquier caso la especificación debe ser absolutamente precisa a la hora de describir qué debe hacer el programa.

Programas correctos (II)

En particular, un programa correcto:

- Debe funcionar para todos los casos previstos, dando el resultado esperado según la especificación
- Debe estar completamente libre de errores.
- Debe terminar adecuadamente

Además, la buena metodología de programación busca que los programas:

- Estén bien estructurados (bloques de código bien organizados)
- Sean eficientes en tiempo de ejecución y en consumo de memoria (en general, en consumo de recursos).
- Estén escritos de forma clara y bien documentados. Con ello se consigue que sean más fáciles de entender, reutilizar y mantener.

Desarrollo de programas. Metodología.

1. Análisis y Especificación del problema (¿QUÉ?):
   - extraer información relevante, eliminar ambigüedades del planteamiento
   - identificar los datos de entrada o input y los datos de salida o output y qué hay que hacer

2. Diseño de un algoritmo (¿CÓMO?): precisar los pasos para obtener la solución requerida (output) a partir de la entrada (input).
   - Partir de planteamiento general prescindiendo de detalles (dejar pendientes subproblemas más pequeños). Después abordar estos subproblemas con la misma técnica $\rightsquigarrow$ Diseño descendente o divide y vencerás o aproximación por refinamientos sucesivos.

3. Implementar el algoritmo en un lenguaje concreto (como C#, en nuestro caso). Compilarlo, corregir posibles errores de sintaxis, ...

4. Probar y depurar (test): comprobar el funcionamiento del programa con una batería de ejemplos *intentando cubrir toda la casuística posible.*
   - Otra alternativa verificación formal de la corrección del algoritmo $\rightsquigarrow$ demostración formal (matemática) utilizando la lógica de Hoare, verificadores (semi)-automáticos de programas...

Es habitual completar las 4 fases anteriores e iterar, es decir, volver a la fase 1, repasar la especificación (a veces el diseño del algoritmo o la propia implementación requieren modificar la especificación), adaptar la implementación, etc....

En en el *ciclo de vida de un programa*, puede incluirse la fase 5:

- Mantenimiento: modificaciones y actualizaciones del programa para satisfacer nuevos requisitos o aumentar prestaciones (o corregir errores no detectados).

Ejemplo (I)

Problema:

averiguar el más pequeño entre dos números dados en cualquier orden

Primer paso, **Análisis y especificación**:

- Información irrelevante?: "dados en cualquier orden"
- Ambigüedad?: ¿cuál es el más pequeño entre 6 y 6?...
  - suponemos que el usuario (cliente) desea 6 como respuesta (o se lo preguntamos para aclararlo)
- Imprecisión?: los números dados, ¿son naturales, enteros, reales, complejos?...
  - supondremos que son enteros (o preguntamos al cliente)

Ejemplo (II)

Especificación (distintas formas de hacerla):

- en lenguaje natural (español), pero precisa:

  determinar el mínimo entre dos enteros dados
- más formal, apoyada en lenguaje lógico/matemático:

  dados $x, y \in \mathbb{Z}$ determinar $z = min(x, y)$ siendo

  $$min(x,y) = \begin{cases} x & \text{si } x \leq y \\ y & \text{en otro caso} \end{cases}$$
- completamente formal, p.e., en estilo funcional:
  se nos pide una función $f : (\mathbb{Z}, \mathbb{Z}) \to \mathbb{Z}$ que verifique:

  $$f(x,y) = \begin{cases} x & \text{si } x \leq y \\ y & \text{en otro caso} \end{cases}$$

Identificamos entrada y salida (input/output):

- input: dos números enteros x, y
- output: el más pequeño de ellos, mínimo entre x y y.

Ejemplo (III)

Segundo paso, diseño del algoritmo: secuencia de acciones a realizar.

- Aproximación I:

  solicitar de teclado los números de entrada x e y
  calcular en z el mínimo de x e y
  escribir z en pantalla
- Aproximación II (refinamiento de I):

  solicitar de teclado los números de entrada x e y
  si $x \leq y$ hacer $z = x$
  en caso contrario hacer $z = y$
  escribir z en pantalla

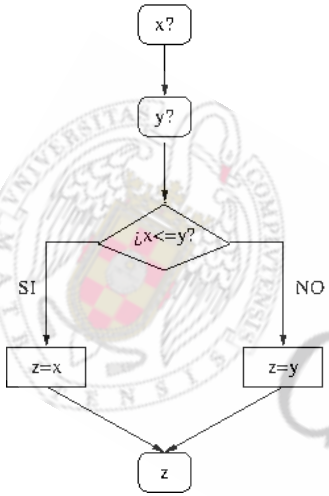
Ejemplo (III)

- Aproximación III (refinamiento de II):

  escribir 'x? ' en pantalla
  leer el valor de x de teclado
  escribir 'y? ' en pantalla
  leer el valor de y de teclado
  si $x \leq y$ hacer $z = x$
  en caso contrario hacer $z = y$
  escribir 'z =' en pantalla
  escribir el valor de z en pantalla

En este nivel de refinamiento se ha detallado suficientemente el algoritmo. Si no, se continuaría refinando.

Diagramas de flujo para algoritmos: otra forma de presentar los algoritmos (no la utilizaremos habitualmente):



Ejemplo (IV)

Ahora habría que hacer la implementación en un lenguaje concreto. Por ejemplo, en Python se podría hacer como (ahora no nos importan los detalles del lenguaje):

```
print("Primer entero: ", end="")
x = int(input())
print("Segundo entero: ", end="")
y = int(input())
if x <= y:
      z = x
else:
      z = y
print("El menor es: ", z)
```


Ejemplo (V)

Quinto paso, **test** (pruebas)

```
Primer entero:  8
Segundo entero:  -17
El menor es:  -17
```

Podemos probar multitud de pares de números
...pero son tests, *no pruebas de corrección*.

Verificación formal: precondición $\xrightarrow{programa}$ postcondición

$\{P_0 : x, y, z \in \mathbb{Z}\}$
```
if x <= y:
    z = x
else:
    z = y
```
$\{P_1 : z = min(x, y)\}$

La **lógica de Hoare** define el comportamiento de cada instrucción y permite demostrar formalmente la corrección del algoritmo.

Ejemplo (VI)

Mantenimiento. El usuario solicita cambios:

- Cambio en la especificación: los números en vez de enteros que sean reales (ampliación de la funcionalidad).
- Cambios de petición y presentación de datos: que pida los dos números a la vez, que escriba en salida también los números de entrada.

En la práctica, en grandes programas a veces hay pequeños (o grandes) errores (bugs). Una parte del mantenimiento también consiste en corregir estos errores.

Esta secuencia es lo que habitualmente se denomina *ciclo de vida del Software*.

Otro ejemplo

Problema:

calcular la suma de los n primeros naturales

- **Análisis y especificación:**
  El 0 cuenta? Qué pasa si $n == 0$?
  Dado $n \in \mathbb{N}$ evaluar $1 + 2 + \ldots + n$
  (en términos matemáticos, calcular $\Sigma_{i=1}^{n} i$)

  input: $n \in \mathbb{N}$; output: $1 + 2 + \ldots + n$

- **Algoritmo:** (sin aplicar la fórmula conocida)
  Aproximación I:
  solicitar de teclado el valor de n
  calcular Suma $= 1 + 2 + \ldots + n$
  escribir Suma en pantalla

Compiladores

Refinamiento del paso de cálculo

inicializar Suma $= 0$
repetir desde $i = 1$ *hasta* n
  $Suma = Suma + i$

Correcto?

Implementación (ahora en Pascal):

```
program sumatorio;    {Este programa calcula la suma...}
  var i, n, suma: integer;
  begin
    write('valor de n: ');
    readln(n);
    suma := 0;              {inicialización del acumulador}
    for i := 1 to n do      {se va incrementando el acumulador}
        suma := suma+i;     {sumando los valores de i}
    writeln('la suma es: ', suma);
  end.
```

Luego depuración, mantenimiento...

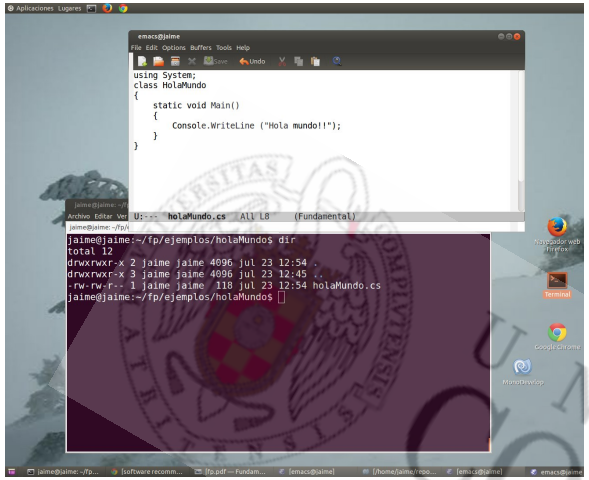
Nuestro primer programa en C#: "hola mundo!"

```
using System;
class HolaMundo
{
    static void Main()
    {
        Console.WriteLine ("Hola mundo!!");
    }
}
```

No es necesario entender este programa por ahora...

- Lo escribimos en nuestro editor de texto favorito (gedit, atom, emacs, Notepad, ...)
- Lo guardamos en un archivo `holaMundo.cs`
- Cómo lo ejecutamos?

Compilando desde línea de comandos (I)

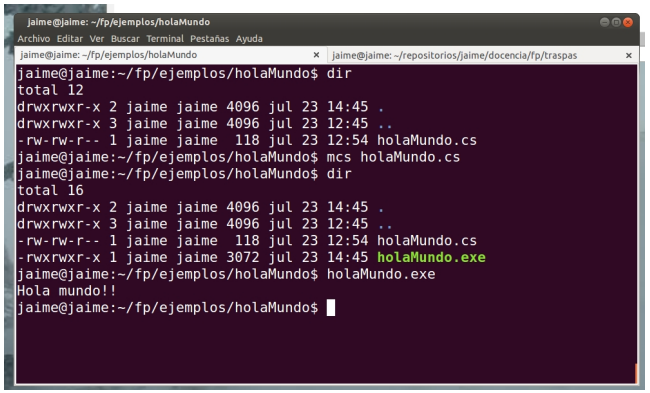


La línea de comandos todavía existe! (en Linux, Windows, Mac...)

En el principio... fue la línea de comandos, Neal Stephenson, 1999,

`http://biblioweb.sindominio.net/telematica/command_es/`

Compilando desde línea de comandos (II)



- Compilación desde línea de comandos (desde un terminal):
  `> mcs holaMundo.cs`
- Esto produce un ejecutable `holaMundo.exe`
- Para ejecutar el programa, desde línea de comandos:
  `> holaMundo.exe`
- Y produce el resultado esperado... acabamos de escribir, compilar y ejecutar nuestro primer programa!

Pero...

- ¿Qué significa exactamente compilar un programa?
- ¿Qué es un compilador?
- ¿Qué significa ejecutar un programa?
- ¿Quién compila?
- ¿Quién ejecuta?

¿Qué significa exactamente compilar un programa?

Compilar es traducir: se traduce el código fuente escrito en un lenguaje de programación (como C#) a código objeto:

- puede ser código binario para una máquina real (CPU) directamente ejecutable por la misma (ceros y unos que "entiende" el ordenador, i.e., instrucciones para el microprocesador)
- o puede ser código para una máquina virtual (bytecode), que puede ser fácilmente convertible en código ejecutable de manera eficiente.

Un compilador es un programa que traduce un programa escrito en un lenguaje de programación a otro lenguaje, para poder ejecutarlo en el ordenador.

En concreto, C# se compila/traduce a un lenguaje intermedio para máquina virtual (Common Intermediate Language CIL), utilizado en la plataforma .NET (usamos el compilador **mcs**).

¿Qué significa ejecutar un programa? ¿Quién ejecuta?

- Ejecutar un programa es hacerlo funcionar en el ordenador. Para ello, el sistema operativo (linux, windows,...) lo carga en memoria y la CPU efectúa las instrucciones de ese programa.
- El sistema operativo (SO) es el que ejecuta los programas en el ordenador.
- El sistema operativo, a su vez es un programa de base (kernel del SO) que utiliza un conjunto de programas (más o menos básicos).

Lenguajes de programación y compiladores

Las ideas de compilador y lenguaje de programación están muy ligadas, pero no son lo mismo:

- En general, puede haber distintos compiladores para un mismo lenguaje de programación. Por ejemplo, para el lenguaje C# hay varios compiladores como *MS Visual Studio* o *MonoDevelop*.

Otro tipo de programas muy relacionados con los compiladores son los intérpretes. También hacen una traducción de lenguaje fuente a objeto; en este caso el programa se va ejecutando a medida que se hace la traducción, mientras que el compilador hace toda la traducción y genera el código objeto (ejecutable), pero no hace la ejecución misma.

Entornos de desarrollo

En la actualidad es muy frecuente que los compiladores se distribuyan como parte de entornos integrados de desarrollo: entorno gráfico para el desarrollo de programas que incluye:

- Editor de texto: con resaltado de sintaxis, auto-completado inteligente de código, herramientas de construcción automáticas (plantillas de programa, etc).
- El compilador propiamente dicho no se lanza desde línea de comandos, sino con botones (a golpe de click de ratón).
- Otras herramientas adaptadas al lenguaje concreto: enlazador de librerías, navegador de clases, ayuda sensitiva, etc.
- Depurador de código (debugger). Es una herramienta que permite hacer trazas de ejecución del programa: ejecutar el programa *paso a paso* (seguir el flujo de ejecución) viendo en el contenido de las variables, la pila de ejecución, etc.

  Es un gran aliado para el programador para detectar y corregir errores.

Entorno de desarrollo para C#

Utilizaremos el entorno *Microsoft Visual Studio* (MSVC) (el mismo que en la asignatura de Motores):

- Accesible en `https://visualstudio.microsoft.com/es/`
- Versión gratuita para particulares y uso académico *Community*

Otros entornos de programación (multiplataforma):

- Monodevelop `https://www.monodevelop.com/`
- Visual Studio Code `https://code.visualstudio.com/`

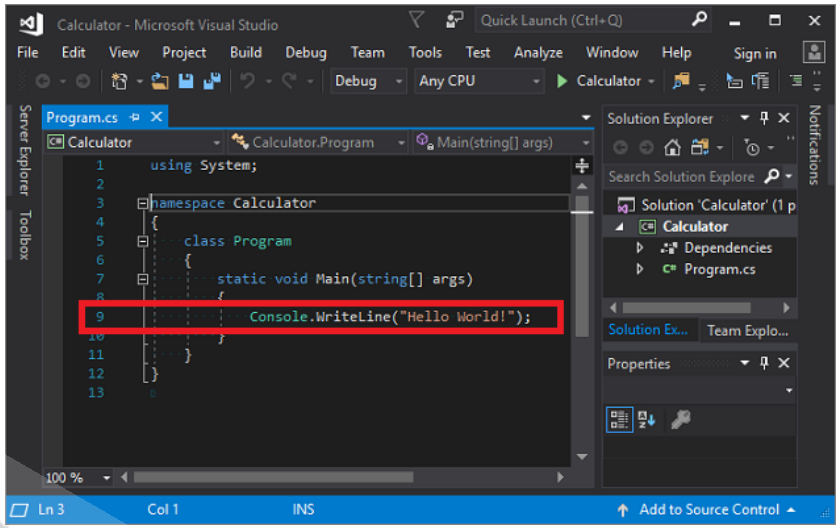
Creación de una aplicación de Consola

- Abrir Visual Studio
- Crear un proyecto nuevo, consola, C#
- Darle nombre
- Darle ubicación (lugar de almacenamiento). En los laboratorios: `c:\hlocal\...`

`https://docs.microsoft.com/es-es/visualstudio/get-started/csharp/tutorial-console?view=vs-2019`

Entorno MSVC

El entorno genera automáticamente una plantilla por defecto para nuestro programa:



El entorno Visual Studio para C#

Damos al *play!*

- El entorno lanza el compilador de C#
- Produce un ejecutable
- Y lo ejecuta