**First Assigment: Basic Concepts and Pseudocoding**

* **Algoritmo:** un algoritmo es un conjunto de pasos ordenados.
* **Lenguaje de Programación:** un lenguaje de programación es una manera formal de expresar un algoritmo para que una computadora pueda resolver el problema.
* **Lenguaje de Marcado:** un lenguaje de marcado es una manera de codificar un documento que, además del texto, usa marcas o etiquetas que contienen información sobre la estructura del texto y su presentación.
* **Entorno de Desarrollo Integrado (IDE):** un IDE es una aplicación que provee servicios integrales para facilitar el desarrollo de software. Un IDE usualmente tiene un editor de código fuente, un depurador y herramientas para la automatización de la construcción del proyecto.
* **Variable:** una variable es un objeto el cual puede cambiar de valor. Esta variable está formada por un espacio en la memoria y un nombre simbólico o identificador asociado a ese espacio.
* **Tipos de variables:** las variables de tipo global son aquellas que pueden ser usadas en cualquier parte del código, y las variables de tipo local son aquellas que solo funcionan dentro de una parte específica del código. Normalmente son declaradas dentro de una función.
* **¿Cómo un programa es ejecutado en memoria?**

http://www.itlalaguna.edu.mx/academico/carreras/sistemas/sitemasoperativos2/Alma\_virtual.html

It really depends on the system, but modern OSes with virtual memory tend to load their process images and allocate memory something like this:

+---------+

| stack | function-local variables, return addresses, return values, etc.

| | often grows downward, commonly accessed via "push" and "pop" (but can be

| | accessed randomly, as well; disassemble a program to see)

+---------+

| shared | mapped shared libraries (C libraries, math libs, etc.)

| libs |

+---------+

| hole | unused memory allocated between the heap and stack "chunks", spans the

| | difference between your max and min memory, minus the other totals

+---------+

| heap | dynamic, random-access storage, allocated with 'malloc' and the like.

+---------+

| bss | Uninitialized global variables; must be in read-write memory area

+---------+

| data | data segment, for globals and static variables that are initialized

| | (can further be split up into read-only and read-write areas, with

| | read-only areas being stored elsewhere in ROM on some systems)

+---------+

| text | program code, this is the actual executable code that is running.

+---------+

This is the general process address space on many common virtual-memory systems. The "hole" is the size of your total memory, minus the space taken up by all the other areas; this gives a large amount of space for the heap to grow into. This is also "virtual", meaning it maps to your actual memory through a translation table, and may be actually stored at any location in actual memory. It is done this way to protect one process from accessing another process's memory, and to make each process think it's running on a complete system.

Note that the positions of, e.g., the stack and heap may be in a different order on some systems (see Billy O'Neal's answer below for more details on Win32).

Other systems can be very different. DOS, for instance, ran in real mode, and its memory allocation when running programs looked much differently:

+-----------+ top of memory

| extended | above the high memory area, and up to your total memory; needed drivers to

| | be able to access it.

+-----------+ 0x110000

| high | just over 1MB->1MB+64KB, used by 286s and above.

+-----------+ 0x100000

| upper | upper memory area, from 640kb->1MB, had mapped memory for video devices, the

| | DOS "transient" area, etc. some was often free, and could be used for drivers

+-----------+ 0xA0000

| USER PROC | user process address space, from the end of DOS up to 640KB

+-----------+

|command.com| DOS command interpreter

+-----------+

| DOS | DOS permanent area, kept as small as possible, provided routines for display,

| kernel | \*basic\* hardware access, etc.

+-----------+ 0x600

| BIOS data | BIOS data area, contained simple hardware descriptions, etc.

+-----------+ 0x400

| interrupt | the interrupt vector table, starting from 0 and going to 1k, contained

| vector | the addresses of routines called when interrupts occurred. e.g.

| table | interrupt 0x21 checked the address at 0x21\*4 and far-jumped to that

| | location to service the interrupt.

+-----------+ 0x0

You can see that DOS allowed direct access to the operating system memory, with no protection, which meant that user-space programs could generally directly access or overwrite anything they liked.

In the process address space, however, the programs tended to look similar, only they were described as code segment, data segment, heap, stack segment, etc., and it was mapped a little differently. But most of the general areas were still there.

Upon loading the program and necessary shared libs into memory, and distributing the parts of the program into the right areas, the OS begins executing your process wherever its main method is at, and your program takes over from there, making system calls as necessary when it needs them.

Different systems (embedded, whatever) may have very different architectures, such as stackless systems, Harvard architecture systems (with code and data being kept in separate physical memory), systems which actually keep the BSS in read-only memory (initially set by the programmer), etc. But this is the general gist.

You said:

I also know that a computer program uses two kinds of memory: stack and heap, which are also part of the primary memory of the computer.

"Stack" and "heap" are just abstract concepts, rather than (necessarily) physically distinct "kinds" of memory.

A stack is merely a last-in, first-out data structure. In the x86 architecture, it can actually be addressed randomly by using an offset from the end, but the most common functions are PUSH and POP to add and remove items from it, respectively. It is commonly used for function-local variables (so-called "automatic storage"), function arguments, return addresses, etc. (more below)

A "heap" is just a nickname for a chunk of memory that can be allocated on demand, and is addressed randomly (meaning, you can access any location in it directly). It is commonly used for data structures that you allocate at runtime (in C++, using new and delete, and malloc and friends in C, etc).

The stack and heap, on the x86 architecture, both physically reside in your system memory (RAM), and are mapped through virtual memory allocation into the process address space as described above.

The registers (still on x86), physically reside inside the processor (as opposed to RAM), and are loaded by the processor, from the TEXT area (and can also be loaded from elsewhere in memory or other places depending on the CPU instructions that are actually executed). They are essentially just very small, very fast on-chip memory locations that are used for a number of different purposes.

Register layout is highly dependent on the architecture (in fact, registers, the instruction set, and memory layout/design, are exactly what is meant by "architecture"), and so I won't expand upon it, but recommend you take an assembly language course to understand them better.

Your question:

At what point is the stack used for the execution of the instructions? Instructions go from the RAM, to the stack, to the registers?

The stack (in systems/languages that have and use them) is most often used like this:

int mul( int x, int y ) {

return x \* y; // this stores the result of MULtiplying the two variables

// from the stack into the return value address previously

// allocated, then issues a RET, which resets the stack frame

// based on the arg list, and returns to the address set by

// the CALLer.

}

int main() {

int x = 2, y = 3; // these variables are stored on the stack

mul( x, y ); // this pushes y onto the stack, then x, then a return address,

// allocates space on the stack for a return value,

// then issues an assembly CALL instruction.

}

Write a simple program like this, and then compile it to assembly (gcc -S foo.c if you have access to GCC), and take a look. The assembly is pretty easy to follow. You can see that the stack is used for function local variables, and for calling functions, storing their arguments and return values. This is also why when you do something like:

f( g( h( i ) ) );

All of these get called in turn. It's literally building up a stack of function calls and their arguments, executing them, and then popping them off as it winds back down (or up ;). However, as mentioned above, the stack (on x86) actually resides in your process memory space (in virtual memory), and so it can be manipulated directly; it's not a separate step during execution (or at least is orthogonal to the process).

FYI, the above is the [C calling convention](http://en.wikipedia.org/wiki/X86_calling_conventions#cdecl), also used by C++. Other languages/systems may push arguments onto the stack in a different order, and some languages/platforms don't even use stacks, and go about it in different ways.

Also note, these aren't actual lines of C code executing. The compiler has converted them into machine language instructions in your executable. ~~They are then (generally) copied from the TEXT area into the CPU pipeline, then into the CPU registers, and executed from there.~~ **[This was incorrect. See**[**Ben Voigt's correction**](http://stackoverflow.com/questions/5162580/what-happens-when-a-computer-program-runs/5163892#5163892)**below.]**

http://duartes.org/gustavo/blog/post/anatomy-of-a-program-in-memory/

* **¿Qué representa una variable en RAM?**

https://www.google.com/url?sa=t&rct=j&q=&esrc=s&source=web&cd=4&cad=rja&uact=8&ved=0ahUKEwjBqbKGypvTAhXFShQKHS4uAQ8QFgg1MAM&url=http%3A%2F%2Fwww.colxi.info%2Fdiferencias-entre-la-asignacion-de-memoria-en-el-stack-y-en-el-heap-c%2F&usg=AFQjCNEW97HiewK9lYa\_YUPWj\_6Qtep-Zg&sig2=7LnbX8NsmBmDKkag\_DpmBw

* **Tipos de datos:**

*Char*: las variables de tipo *char* son caracteres. Su tamaño es de 1 byte.

*Short*: las variables de tipo *short* son enteros pequeños. Su tamaño es de 2 bytes y su rango es [-32768,32767]

*Int*: las variables de tipo *int* son enteros. Su tamaño es de 4 bytes y su rango es de

[-231,231 – 1]

*Long*: las variables de tipo *long* son enteros grandes. Su tamaño es de 8 bytes y su rango es de [-263,263-1]

*Float*: las variables de tipo *float* son variables reales. Su tamaño de 4 bytes y su rango es de [±3,4·10-38; ±3,4·1038]

*Double*: las variables de tipo *double* son variables reales grandes. Su tamaño es de 8 bytes y su rango es de [±1,7·10-308.. ±1,7·10308]

*Boolean*: los tipos de datos *boolean* son variables que solo pueden tener uno de estos dos valores: verdadero o falso. Su tamaño es de 1 byte.

Además de los datos básicos o simples, se pueden construir otros datos a partir de éstos, y se obtienen los datos compuestos o datos agregados, tales como **estructuras, uniones, enumeraciones** (*subrango*, como caso particular de las enumeraciones, al igual de lo que sucede en Pascal), **vectores** o **matrices/tablas** y **cadenas “arrays o arreglos”**; también existen otros datos especiales en lenguajes como C y C++, denominados **punteros (apuntadores)** y **referencias**.

* **Operadores y tipos de operadores:** los operadores son conectores que sirven para armar expresiones aritméticas o lógicas dependiendo del tipo de operadores que se usen. Los operadores se clasifican en aritméticos y lógicos.
* **Programación modular:** La programación modular es un paradigma de programación que consiste en dividir un programa en módulos o subprogramas con el fin de hacerlo más legible y manejable.

Se presenta históricamente como una evolución de la programación estructurada para solucionar problemas de programación más grandes y complejos de lo que esta puede resolver.

Al aplicar la programación modular, un problema complejo debe ser dividido en varios subproblemas más simples, y estos a su vez en otros subproblemas más simples. Esto debe hacerse hasta obtener subproblemas lo suficientemente simples como para poder ser resueltos fácilmente con algún lenguaje de programación. Esta técnica se llama refinamiento sucesivo, divide y vencerás ó análisis descendente (Top-Down).

* **Programación estructurada:** la programación estructurada es un paradigma de programación orientado a mejorar la claridad, calidad y tiempo de desarrollo de un programa de computadora, utilizando únicamente subrutinas y tres estructuras: secuencia, selección (if y switch) e iteración (bucles for y while), considerando innecesario y contraproducente el uso de la instrucción de transferencia incondicional (GOTO), que podría conducir a "código espagueti", que es mucho más difícil de seguir y de mantener, y era la causa de muchos errores de programación.
* **Orquestación:** la orquestación es la organización automatizada, la coordinación y la administración de sistemas informáticos, middleware y servicios.
* **Arquitectura de soluciones:** es el diseño una solución, mapeando los requerimientos funcionales hacia tecnologías. Al hacer el diseño de la arquitectura de solución se crea un puente entre el cliente con sus necesidades, y los ingenieros que implementarán la solución. Esto quiere decir que si algún requerimiento cambia, el arquitecto debe ser capaz de saber qué componentes de la solución son afectados y de qué manera.

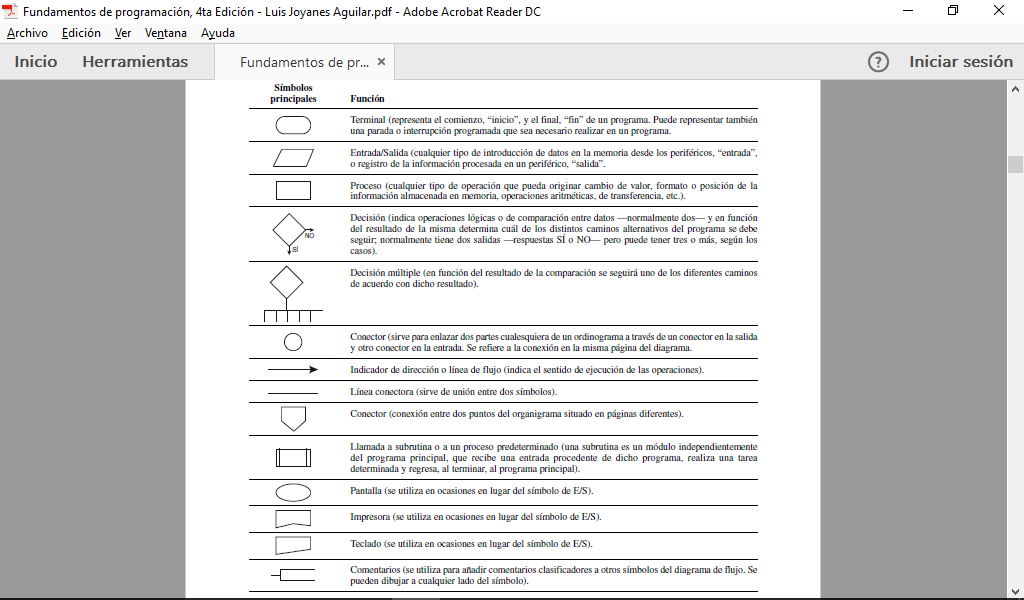
Dentro del diseño se deben tomar en cuenta posibles cambios en el futuro y a las posibles integraciones que la solución pueda requerir. Y entre más compleja sea la solución o más integraciones tenga, más evidente se hace la necesidad de tener un diseño claro para poder implementar la solución.

* **Pseudocódigo:** el pseudocódigo es un lenguaje de especificación (descripción) de algoritmos. El uso de tal lenguaje hace el paso de codificación final (esto es, la traducción a un lenguaje de programación) relativamente fácil.

La ventaja del pseudocódigo es que en su uso, en la planificación de un programa, el programador se puede concentrar en la lógica y en las estructuras de control y no preocuparse de las reglas de un lenguaje específico. Es también fácil modificar el pseudocódigo si se descubren errores o anomalías en la lógica del programa, mientras que en muchas ocasiones suele ser difícil el cambio en la lógica, una vez que está

codificado en un lenguaje de programación.

* **Diagrama de flujo:** un diagrama de flujo (flowchart) es una de las técnicas de representación de algoritmos más antigua y a la vez más utilizada, aunque su empleo ha disminuido considerablemente, sobre todo, desde la aparición de lenguajes de programación estructurados. Un diagrama de flujo es un diagrama que utiliza los símbolos (cajas) estándar y que tiene los pasos de algoritmo escritos en esas cajas unidas por flechas, denominadas líneas de flujo, que indican la secuencia en que se debe ejecutar.
* **Símbolos de Diagrama de flujo**

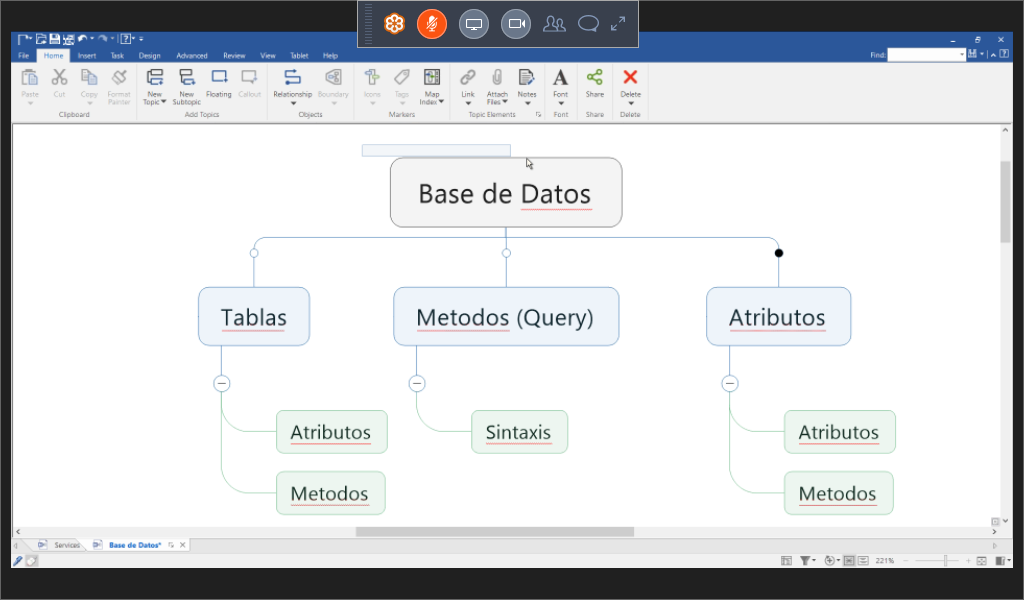


* **¿Qué es una Base de Datos?**

Una base de datos o banco de datos es un conjunto de datos pertenecientes a un mismo contexto y almacenados sistemáticamente para su posterior uso.

* **¿Qué es un query?**

Es una consulta. Un query string es un término informático que se utiliza para hacer referencia a una interacción con una base de datos. Es la parte de una URL que contiene los datos que deben pasar a aplicaciones web.



* **Modelo de datos y tipos de modelos de datos:** un modelo de base de datos es un tipo de modelo de datos que determina la estructura lógica de una base de datos y de manera fundamental determina el modo de almacenar, organizar y manipular los datos.

Entre los modelos lógicos comunes para bases de datos se encuentran:

* + Modelo jerárquico
  + Modelo en red
  + Modelo relacional
  + Modelo entidad–relación
  + Modelo entidad–relación extendido
  + modelo de objetos
  + modelo documental
  + Modelo entidad–atributo–valor
  + modelo en estrella

Los modelos físicos de datos incluyen:

* + índice invertido
  + fichero plano

Otros modelos lógicos pueden ser:

* + modelo asociativo
  + modelo multidimensional
  + modelo multivalor
  + modelo semántico
  + base de datos XML
  + grafo etiquetado
  + Triplestore
* **Modelo E-R:** El modelo de datos entidad-relación está basado en una percepción del mundo real que consta de una colección de objetos básicos, llamados entidades, y de relaciones entre esos objetos.

Entidad

Representa una “cosa”, "objeto" o "concepto" del mundo real con existencia independiente, es decir, se diferencia únicamente de otro objeto o cosa, incluso siendo del mismo tipo, o una misma entidad.

Algunos Ejemplos:

* + Una persona. (Se diferencia de cualquier otra persona, incluso siendo gemelos).
  + Un automóvil. (Aunque sean de la misma marca, el mismo modelo,..., tendrán atributos diferentes, por ejemplo, el número de chasis).
  + Una casa (Aunque sea exactamente igual a otra, aún se diferenciará en su dirección).

Una entidad puede ser un objeto con existencia física como: una persona, un animal, una casa, etc. (entidad concreta); o un objeto con existencia conceptual como: un puesto de trabajo, una asignatura de clases, un nombre, etc. (entidad abstracta).

Una entidad está descrita y se representa por sus características o atributos. Por ejemplo, la entidad Persona las características: Nombre, Apellido, Género, Estatura, Peso, Fecha de nacimiento.

Atributos

Los atributos son las características que definen o identifican a una entidad. Estas pueden ser muchas, y el diseñador solo utiliza o implementa las que considere más relevantes.

En un conjunto de entidades del mismo tipo, cada entidad tiene valores específicos asignados para cada uno de sus atributos, de esta forma, es posible su identificación unívoca.

Ejemplos:

A la colección de entidades «alumnos», con el siguiente conjunto de atributos en común, (id, nombre, edad, semestre), pertenecen las entidades:

* + (1, Sophia, 15 años, 2)
  + (2, Josefa, 19 años, 5)
  + (3, Carlos, 20 años, 2)

Cada una de las entidades pertenecientes a este conjunto se diferencia de las demás por el valor de sus atributos. Nótese que dos o más entidades diferentes pueden tener los mismos valores para algunos de sus atributos, pero nunca para todos.

En particular, los atributos identificativos son aquellos que permiten diferenciar a una instancia de la entidad de otra distinta. Por ejemplo, el atributo identificativo que distingue a un alumno de otro es su número de id.

Para cada atributo, existe un dominio del mismo, este hace referencia al tipo de datos que será almacenado a restricciones en los valores que el atributo puede tomar (cadenas de caracteres, números, solo dos letras, solo números mayores que cero, solo números enteros...).

Cuando algún atributo correspondiente a una entidad no tiene un valor determinado, recibe el valor nulo, bien sea porque no se conoce, porque no existe o porque no se sabe nada al respecto del mismo.

Conjunto de relaciones

Consiste en una colección, o conjunto, de relaciones de la misma naturaleza.

Ejemplo:

Dados los conjuntos de entidades "Habitación" y "Huésped", todas las relaciones de la forma habitación-huésped, permiten obtener la información de los huéspedes y sus respectivas habitaciones.

La dependencia o asociación entre los conjuntos de entidades es llamada participación. En el ejemplo anterior los conjuntos de entidades "Habitación" y "Huésped" participan en el conjunto de relaciones habitación-huésped.

Se llama grado del conjunto de relaciones a la cantidad de conjuntos de entidades participantes en la relación.