<https://www.tutorialspoint.com/dbms/er_model_basic_concepts.htm>

Data models define how the logical structure of a database is modeled. Data Models are fundamental entities to introduce abstraction in a DBMS. Data models define how data is connected to each other and how they are processed and stored inside the system.

The very first data model could be flat data-models, where all the data used are to be kept in the same plane. Earlier data models were not so scientific, hence they were prone to introduce lots of duplication and update anomalies.

Entity-Relationship Model

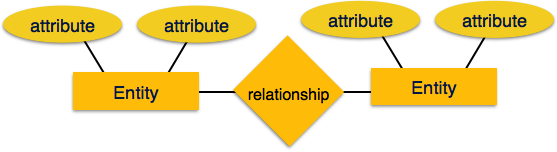
Entity-Relationship (ER) Model is based on the notion of real-world entities and relationships among them. While formulating real-world scenario into the database model, the ER Model creates entity set, relationship set, general attributes and constraints.

ER Model is best used for the conceptual design of a database.

ER Model is based on −

* **Entities** and their *attributes.*
* **Relationships** among entities.

These concepts are explained below.



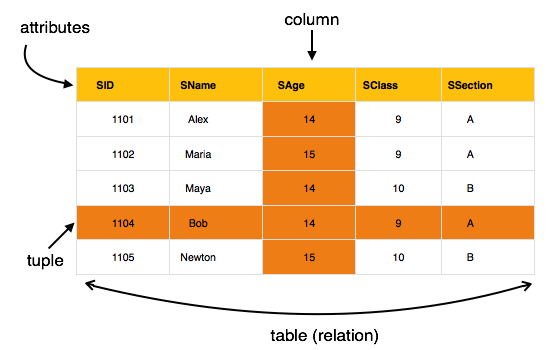
* **Entity** − An entity in an ER Model is a real-world entity having properties called **attributes**. Every **attribute** is defined by its set of values called **domain**. For example, in a school database, a student is considered as an entity. Student has various attributes like name, age, class, etc.
* **Relationship** − The logical association among entities is called ***relationship***. Relationships are mapped with entities in various ways. Mapping cardinalities define the number of association between two entities.

Mapping cardinalities −

* + one to one
  + one to many
  + many to one
  + many to many

Relational Model

The most popular data model in DBMS is the Relational Model. It is more scientific a model than others. This model is based on first-order predicate logic and defines a table as an **n-ary relation**.

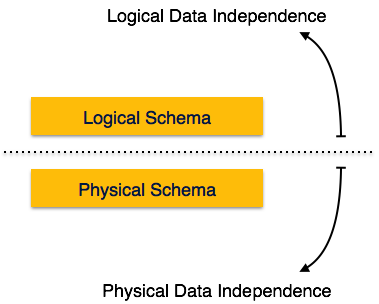


The main highlights of this model are −

* Data is stored in tables called **relations**.
* Relations can be normalized.
* In normalized relations, values saved are atomic values.
* Each row in a relation contains a unique value.
* Each column in a relation contains values from a same domain.

## Data Independence

A database system normally contains a lot of data in addition to users’ data. For example, it stores data about data, known as metadata, to locate and retrieve data easily. It is rather difficult to modify or update a set of metadata once it is stored in the database. But as a DBMS expands, it needs to change over time to satisfy the requirements of the users. If the entire data is dependent, it would become a tedious and highly complex job.



Metadata itself follows a layered architecture, so that when we change data at one layer, it does not affect the data at another level. This data is independent but mapped to each other.

## Logical Data Independence

Logical data is data about database, that is, it stores information about how data is managed inside. For example, a table (relation) stored in the database and all its constraints, applied on that relation.

Logical data independence is a kind of mechanism, which liberalizes itself from actual data stored on the disk. If we do some changes on table format, it should not change the data residing on the disk.

## Physical Data Independence

All the schemas are logical, and the actual data is stored in bit format on the disk. Physical data independence is the power to change the physical data without impacting the schema or logical data.

For example, in case we want to change or upgrade the storage system itself − suppose we want to replace hard-disks with SSD − it should not have any impact on the logical data or schemas.

The ER model defines the conceptual view of a database. It works around real-world entities and the associations among them. At view level, the ER model is considered a good option for designing databases.

Entity

An entity can be a real-world object, either animate or inanimate, that can be easily identifiable. For example, in a school database, students, teachers, classes, and courses offered can be considered as entities. All these entities have some attributes or properties that give them their identity.

An entity set is a collection of similar types of entities. An entity set may contain entities with attribute sharing similar values. For example, a Students set may contain all the students of a school; likewise a Teachers set may contain all the teachers of a school from all faculties. Entity sets need not be disjoint.

Attributes

Entities are represented by means of their properties, called **attributes**. All attributes have values. For example, a student entity may have name, class, and age as attributes.

There exists a domain or range of values that can be assigned to attributes. For example, a student's name cannot be a numeric value. It has to be alphabetic. A student's age cannot be negative, etc.

Types of Attributes

* **Simple attribute** − Simple attributes are atomic values, which cannot be divided further. For example, a student's phone number is an atomic value of 10 digits.
* **Composite attribute** − Composite attributes are made of more than one simple attribute. For example, a student's complete name may have first\_name and last\_name.
* **Derived attribute** − Derived attributes are the attributes that do not exist in the physical database, but their values are derived from other attributes present in the database. For example, average\_salary in a department should not be saved directly in the database, instead it can be derived. For another example, age can be derived from data\_of\_birth.
* **Single-value attribute** − Single-value attributes contain single value. For example − Social\_Security\_Number.
* **Multi-value attribute** − Multi-value attributes may contain more than one values. For example, a person can have more than one phone number, email\_address, etc.

These attribute types can come together in a way like −

* simple single-valued attributes
* simple multi-valued attributes
* composite single-valued attributes
* composite multi-valued attributes

Entity-Set and Keys

Key is an attribute or collection of attributes that uniquely identifies an entity among entity set.

For example, the roll\_number of a student makes him/her identifiable among students.

* **Super Key** − A set of attributes (one or more) that collectively identifies an entity in an entity set.
* **Candidate Key** − A minimal super key is called a candidate key. An entity set may have more than one candidate key.
* **Primary Key** − A primary key is one of the candidate keys chosen by the database designer to uniquely identify the entity set.

Relationship

The association among entities is called a relationship. For example, an employee **works\_at** a department, a student **enrolls** in a course. Here, Works\_at and Enrolls are called relationships.

Relationship Set

A set of relationships of similar type is called a relationship set. Like entities, a relationship too can have attributes. These attributes are called **descriptive attributes**.

Degree of Relationship

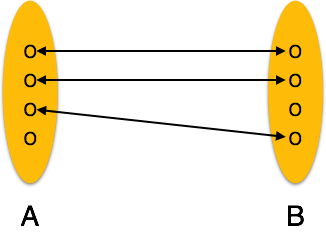
The number of participating entities in a relationship defines the degree of the relationship.

* Binary = degree 2
* Ternary = degree 3
* n-ary = degree

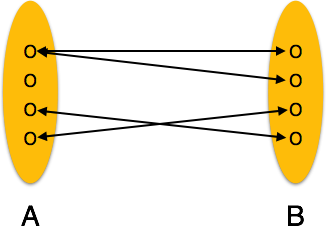
Mapping Cardinalities

**Cardinality** defines the number of entities in one entity set, which can be associated with the number of entities of other set via relationship set.

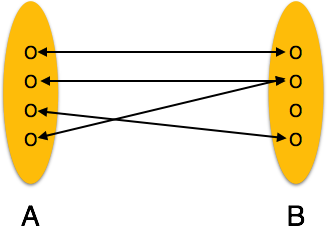
* **One-to-one** − One entity from entity set A can be associated with at most one entity of entity set B and vice versa.



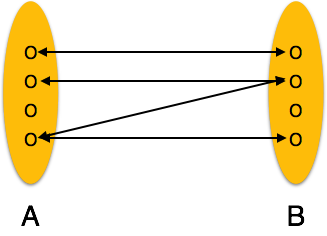
* **One-to-many** − One entity from entity set A can be associated with more than one entities of entity set B however an entity from entity set B, can be associated with at most one entity.



* **Many-to-one** − More than one entities from entity set A can be associated with at most one entity of entity set B, however an entity from entity set B can be associated with more than one entity from entity set A.



* **Many-to-many** − One entity from A can be associated with more than one entity from B and vice versa.



Let us now learn how the ER Model is represented by means of an ER diagram. Any object, for example, entities, attributes of an entity, relationship sets, and attributes of relationship sets, can be represented with the help of an ER diagram.

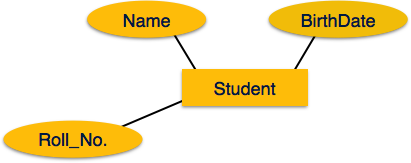
Entity

Entities are represented by means of rectangles. Rectangles are named with the entity set they represent.

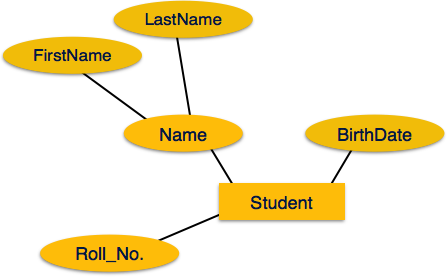
Entities in a school database

Attributes

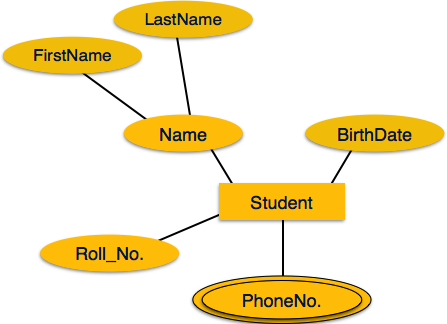
Attributes are the properties of entities. Attributes are represented by means of ellipses. Every ellipse represents one attribute and is directly connected to its entity (rectangle).



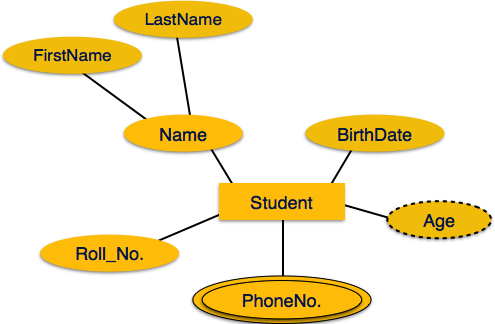
If the attributes are **composite**, they are further divided in a tree like structure. Every node is then connected to its attribute. That is, composite attributes are represented by ellipses that are connected with an ellipse.



**Multivalued** attributes are depicted by double ellipse.



**Derived** attributes are depicted by dashed ellipse.



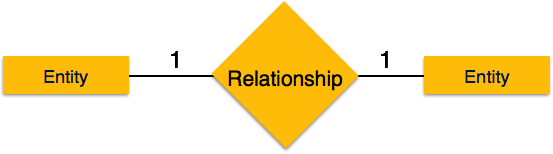
Relationship

Relationships are represented by diamond-shaped box. Name of the relationship is written inside the diamond-box. All the entities (rectangles) participating in a relationship, are connected to it by a line.

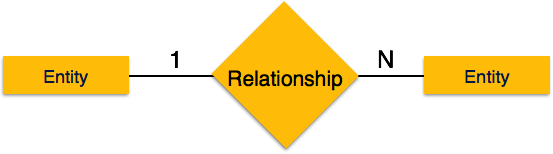
Binary Relationship and Cardinality

A relationship where two entities are participating is called a **binary relationship**. Cardinality is the number of instance of an entity from a relation that can be associated with the relation.

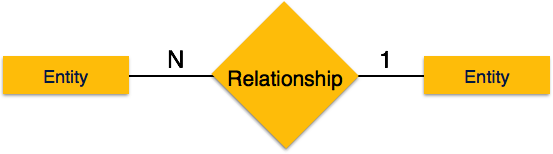
* **One-to-one** − When only one instance of an entity is associated with the relationship, it is marked as '1:1'. The following image reflects that only one instance of each entity should be associated with the relationship. It depicts one-to-one relationship.



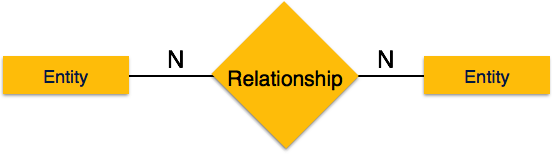
* **One-to-many** − When more than one instance of an entity is associated with a relationship, it is marked as '1:N'. The following image reflects that only one instance of entity on the left and more than one instance of an entity on the right can be associated with the relationship. It depicts one-to-many relationship.



* **Many-to-one** − When more than one instance of entity is associated with the relationship, it is marked as 'N:1'. The following image reflects that more than one instance of an entity on the left and only one instance of an entity on the right can be associated with the relationship. It depicts many-to-one relationship.

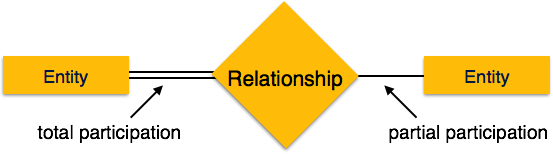


* **Many-to-many** − The following image reflects that more than one instance of an entity on the left and more than one instance of an entity on the right can be associated with the relationship. It depicts many-to-many relationship.



Participation Constraints

* **Total Participation** − Each entity is involved in the relationship. Total participation is represented by double lines.
* **Partial participation** − Not all entities are involved in the relationship. Partial participation is represented by single lines.

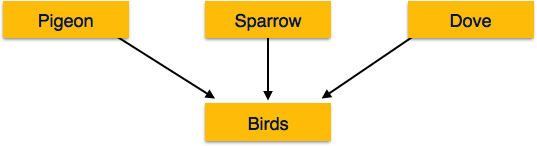


The ER Model has the power of expressing database entities in a conceptual hierarchical manner. As the hierarchy goes up, it generalizes the view of entities, and as we go deep in the hierarchy, it gives us the detail of every entity included.

Going up in this structure is called **generalization**, where entities are clubbed together to represent a more generalized view. For example, a particular student named Mira can be generalized along with all the students. The entity shall be a student, and further, the student is a person. The reverse is called **specialization** where a person is a student, and that student is Mira.

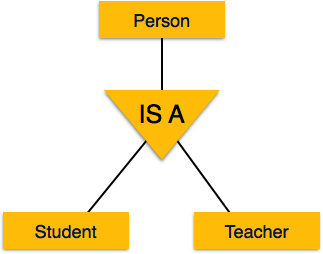
## Generalization

As mentioned above, the process of generalizing entities, where the generalized entities contain the properties of all the generalized entities, is called generalization. In generalization, a number of entities are brought together into one generalized entity based on their similar characteristics. For example, pigeon, house sparrow, crow and dove can all be generalized as Birds.



## Specialization

Specialization is the opposite of generalization. In specialization, a group of entities is divided into sub-groups based on their characteristics. Take a group ‘Person’ for example. A person has name, date of birth, gender, etc. These properties are common in all persons, human beings. But in a company, persons can be identified as employee, employer, customer, or vendor, based on what role they play in the company.

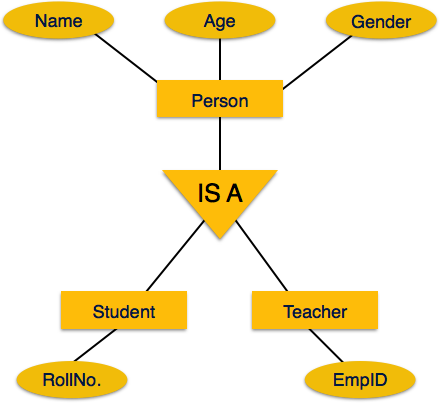


Similarly, in a school database, persons can be specialized as teacher, student, or a staff, based on what role they play in school as entities.

## Inheritance

We use all the above features of ER-Model in order to create classes of objects in object-oriented programming. The details of entities are generally hidden from the user; this process known as **abstraction**.

Inheritance is an important feature of Generalization and Specialization. It allows lower-level entities to inherit the attributes of higher-level entities.



For example, the attributes of a Person class such as name, age, and gender can be inherited by lower-level entities such as Student or Teacher.

# **DBMS - Normalization**

Functional Dependency

Functional dependency (FD) is a set of constraints between two attributes in a relation. Functional dependency says that if two tuples have same values for attributes A1, A2,..., An, then those two tuples must have to have same values for attributes B1, B2, ..., Bn.

Functional dependency is represented by an arrow sign (→) that is, X→Y, where X functionally determines Y. The left-hand side attributes determine the values of attributes on the right-hand side.

Armstrong's Axioms

If F is a set of functional dependencies then the closure of F, denoted as F+, is the set of all functional dependencies logically implied by F. Armstrong's Axioms are a set of rules, that when applied repeatedly, generates a closure of functional dependencies.

* **Reflexive rule** − If alpha is a set of attributes and beta is\_subset\_of alpha, then alpha holds beta.
* **Augmentation rule** − If a → b holds and y is attribute set, then ay → by also holds. That is adding attributes in dependencies, does not change the basic dependencies.
* **Transitivity rule** − Same as transitive rule in algebra, if a → b holds and b → c holds, then a → c also holds. a → b is called as a functionally that determines b.

Trivial Functional Dependency

* **Trivial** − If a functional dependency (FD) X → Y holds, where Y is a subset of X, then it is called a trivial FD. Trivial FDs always hold.
* **Non-trivial** − If an FD X → Y holds, where Y is not a subset of X, then it is called a non-trivial FD.
* **Completely non-trivial** − If an FD X → Y holds, where x intersect Y = Φ, it is said to be a completely non-trivial FD.

Normalization

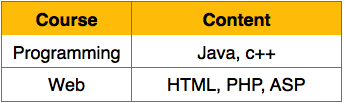
If a database design is not perfect, it may contain anomalies, which are like a bad dream for any database administrator. Managing a database with anomalies is next to impossible.

* **Update anomalies** − If data items are scattered and are not linked to each other properly, then it could lead to strange situations. For example, when we try to update one data item having its copies scattered over several places, a few instances get updated properly while a few others are left with old values. Such instances leave the database in an inconsistent state.
* **Deletion anomalies** − We tried to delete a record, but parts of it was left undeleted because of unawareness, the data is also saved somewhere else.
* **Insert anomalies** − We tried to insert data in a record that does not exist at all.

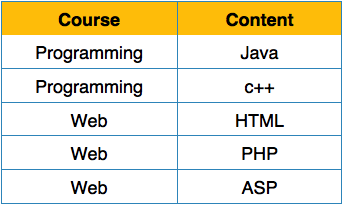
Normalization is a method to remove all these anomalies and bring the database to a consistent state.

First Normal Form

First Normal Form is defined in the definition of relations (tables) itself. This rule defines that all the attributes in a relation must have atomic domains. The values in an atomic domain are indivisible units.



We re-arrange the relation (table) as below, to convert it to First Normal Form.



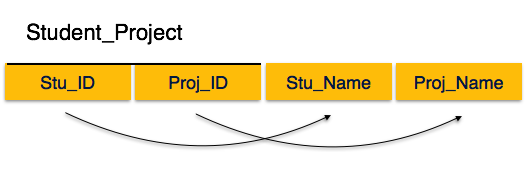
Each attribute must contain only a single value from its pre-defined domain.

Second Normal Form

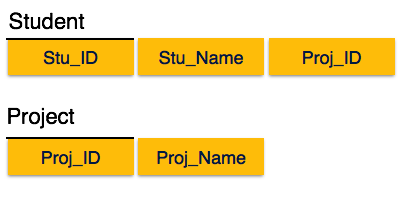
Before we learn about the second normal form, we need to understand the following −

* **Prime attribute** − An attribute, which is a part of the candidate-key, is known as a prime attribute.
* **Non-prime attribute** − An attribute, which is not a part of the prime-key, is said to be a non-prime attribute.

If we follow second normal form, then every non-prime attribute should be fully functionally dependent on prime key attribute. That is, if X → A holds, then there should not be any proper subset Y of X, for which Y → A also holds true.



We see here in Student\_Project relation that the prime key attributes are Stu\_ID and Proj\_ID. According to the rule, non-key attributes, i.e. Stu\_Name and Proj\_Name must be dependent upon both and not on any of the prime key attribute individually. But we find that Stu\_Name can be identified by Stu\_ID and Proj\_Name can be identified by Proj\_ID independently. This is called **partial dependency**, which is not allowed in Second Normal Form.



We broke the relation in two as depicted in the above picture. So there exists no partial dependency.

Third Normal Form

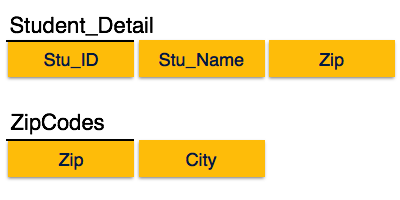
For a relation to be in Third Normal Form, it must be in Second Normal form and the following must satisfy −

* No non-prime attribute is transitively dependent on prime key attribute.
* For any non-trivial functional dependency, X → A, then either −
  + X is a superkey or,
  + A is prime attribute.



We find that in the above Student\_detail relation, Stu\_ID is the key and only prime key attribute. We find that City can be identified by Stu\_ID as well as Zip itself. Neither Zip is a superkey nor is City a prime attribute. Additionally, Stu\_ID → Zip → City, so there exists **transitive dependency**.

To bring this relation into third normal form, we break the relation into two relations as follows −



Boyce-Codd Normal Form

Boyce-Codd Normal Form (BCNF) is an extension of Third Normal Form on strict terms. BCNF states that −

* For any non-trivial functional dependency, X → A, X must be a super-key.

In the above image, Stu\_ID is the super-key in the relation Student\_Detail and Zip is the super-key in the relation ZipCodes. So,

Stu\_ID → Stu\_Name, Zip

and

Zip → City

Which confirms that both the relations are in BCNF

# Storage of Null Values

A null is the absence of a value in a column. Nulls indicate missing, unknown, or

inapplicable data.

Nulls are stored in the database if they fall between columns with data values. In these

cases, they require 1 byte to store the length of the column (zero). Trailing nulls in a

row require no storage because a new row header signals that the remaining columns

in the previous row are null. For example, if the last three columns of a table are null,

then no data is stored for these columns.

**NULLS FIRST | NULLS LAST** Specify whether returned rows containing null values

should appear first or last in the ordering sequence.

NULLS LAST is the default for ascending order, and NULLS FIRST is the default for

descending order.

# **Unix Nohup: Run a Command or Shell-Script Even after You Logout**

*by* SATHIYAMOORTHY

When you execute a Unix job in the background ( using &, bg command), and logout from the session, your process will get killed. You can avoid this using several methods — executing the job with nohup, or making it as batch job using at, batch or cron command.  
  
This quick tip is for beginners. If you’ve been using nohup for a while, leave us a comment and tell us under what situations you use nohup.

In this quick tip, let us review how to make your process running even after you logout, using nohup.

Nohup stands for no hang up, which can be executed as shown below.

nohup syntax:

# nohup command-with-options &

Nohup is very helpful when you have to execute a shell-script or command that take a long time to finish. In that case, you don’t want to be connected to the shell and waiting for the command to complete. Instead, execute it with nohup, exit the shell and continue with your other work.

### Explanation about nohup.out file

By default, the standard output will be redirected to nohup.out file in the current directory. And the standard error will be redirected to stdout, thus it will also go to nohup.out. So, your nohup.out will contain both standard output and error messages from the script that you’ve executed using nohup command.

Instead of using nohup.out, you can also redirect the output to a file using the normal shell redirections.

### Example: Printing lines to both standard output & standard error

while(true)

do

echo "standard output"

echo "standard error" 1>&2

sleep 1;

done

### Execute the script without redirection

$ nohup sh custom-script.sh &

[1] 12034

$ nohup: ignoring input and appending output to `nohup.out'

$ tail -f nohup.out

standard output

standard error

standard output

standard error

..

### Execute the script with redirection

$ nohup sh custom-script.sh > custom-out.log &

[1] 11069

$ nohup: ignoring input and redirecting stderr to stdout

$ tail -f custom-out.log

standard output

standard error

standard output

standard error

..

If you log-out of the shell and login again, you’ll still see the custom-script.sh running in the background.

$ ps aux | grep sathiya

sathiya 12034 0.0 0.1 4912 1080 pts/2 S 14:10 0:00 sh custom-script.sh

# **What are near, far and huge pointers?**

These are some old concepts used in 16 bit intel architectures in the days of MS DOS, not much useful anymore.

**Near pointer** is used to store 16 bit addresses means within current segment on a 16 bit machine. The limitation is that we can only access 64kb of data at a time.

**A far pointer** is typically 32 bit that can access memory outside current segment.  To use this, compiler allocates a segment register to store segment address, then another register to store offset within current segment.

Like far pointer, **huge pointer** is also typically 32 bit and can access outside segment. In case of far pointers, a segment is fixed. In far pointer, the segment part cannot be modified, but in Huge it can be.

### difference between far and huge pointers

as we know by default the pointers are near for example **int \*p** is a near pointer... size of near pointer is 2 bytes in case of 16 bit compiler........ n we already know very well size varies compiler to compiler...... They only store the offset of the address the pointer is referencing. . An address consisting of only an offset has a range of 0 - 64K bytes.... i think there is no need to discuss near pointers anymore.... so come to the main point..... that is far and huge pointers......

**far and huge pointers:**

Far and huge pointers have a size of 4 bytes. They store both the segment and the offset of the address the pointer is referencing. thn what is the difference between them ...........

**Limitation of far pointer:**

We cannot change or modify the segment address of given far address by applying any arithmetic operation on it. That is by using arithmetic operator we cannot jump from one segment to other segment. If you will increment the far address beyond the maximum value of its offset address instead of incrementing segment address it will repeat its offset address in cyclic order. this is also called wrapping.....i.e. if offset is 0xffff and we add 1 then it is 0x0000 and similarly if we decrease 0x0000 by 1 then it is 0xffff and remember there is no change in the segment....

Now i am going to **compare huge and far pointers..........**

1.

**When a far pointer is incremented or decremented ONLY the offset of the pointer is actually incremented or decremented but in case of huge pointer both segment and offset value will change.....**

**like if we have**

int main()

{

char far\* f=(char far\*)0x0000ffff;

printf("%Fp",f+0x1);

return 0;

}

then the output is:

0000:0000

as we see there is no change in segment value.......

and in case of huge........

int main()

{

char huge\* h=(char huge\*)0x0000000f;

printf("%Fp",h+0x1);

return 0;

}

then the o/p is:

0001:0000

it shows bcoz of increment operation not only offset value but segment value also change.......... that means segment will not change in case of far pointers but in case of huge it can move from one segment to another ......

2.

**When relational operators are used on far pointers only the offsets are compared.In other words relational operators will only work on far pointers if the segment values of the pointers being compared are the same. and in case of huge this will not happen, actually comparison of absolute addresses takes place.....**. lets understand with the help of an example...

in far...............................

int main()

{

char far \* p=(char far\*)0x12340001;

char far\* p1=(char far\*)0x12300041;

if(p==p1)

printf("same");

else

printf("different");

return 0;

}

Output:

different

in huge.......................

int main()

{

char huge \* p=(char huge\*)0x12340001;

char huge\* p1=(char huge\*)0x12300041;

if(p==p1)

printf("same");

else

printf("different");

return 0;

}

Output:

same

Explanation:

as we see the absolute address for both p and p1 is 12341 (1234\*10+1 or 1230\*10+41) but they are not consideredequal in 1st case becoz in case of far pointers only offsets are compared i.e. it will check whether 0001==0041.... that we know is false.... and know see what will happen in case of huge.....the comparison operation is performed on absolute addresses that are equal as i already told......

3.

**A far pointer is never noramlized but a huge pointer is normalized . A normalized pointer is one that has as much of the address as possible in the segment, meaning that the offset is never larger than 15.**

suppose if we have 0x1234:1234 then the normalized form of it is 0x1357:0004(absolute address is 13574).......

**A huge pointer is normalized only when some arithmetic operation is performed on it ... and not noramlized during assignment....**

int main()

{

char huge\* h=(char huge\*)0x12341234;

char huge\* h1=(char huge\*)0x12341234;

printf("h=%Fp\nh1=%Fp",h,h1+0x1);

return 0;

}

Output:

h=1234:1234

h1=1357:0005

Explanation:

as i said above huge is not normalized in case of assignment......but if an arithmetic operation is performed on it ..... it will be normalized.... so h is 1234:1234 and h1 is 1357:0005..........that is normalized.......

4.

**The offset of huge pointer is less than 16 because of normalization and not so in case of far pointers...................**

lets take an example to understand what i want to say.....

int main()

{

char far\* f=(char far\*)0x0000000f;

printf("%Fp",f+0x1);

return 0;

}

Output:

0000:0010

in case of huge

int main()

{

char huge\* h=(char huge\*)0x0000000f;

printf("%Fp",h+0x1);

return 0;

}

Output:

0001:0000

Explanation:

as we increment far pointer by 1 it will be 0000:0010...... and as we increment huge pointer by 1 thn it will be 0001:0000 bcoz its offset cant be greater than 15 in other words it will be normalized