***TERM TWO COMPUTER SCIENCE NOTES FOR S5MPC***

## **UNIT 5: DATABASE DESIGN**

## **5.0 INTRODUCTION**

Sometimes ago, organizations usually stored information in a way which was with many risks of being lost, damaged, corrupted and not well organized. It was kept in Traditional File Processing (TFP) which presented many disadvantages.

Nowadays, well-designed database arises to overcome those problems and saves the time in the long run for the user. It is facilitated by a Database Management Systems (DBMS). The design of such database presents 3 main levels called “Database design levels”. These levels help to make easy the understanding and the management of a database. Later on, it will not be very difficult for the software developer to use that database.

## **5.1 DATABASE DESIGN LEVELS**

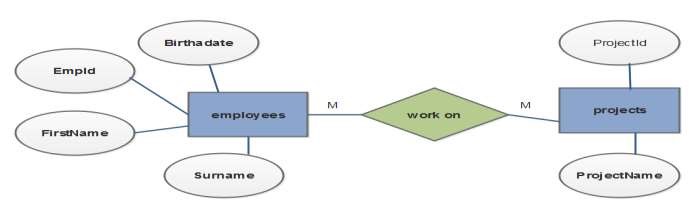
**5.1.1 The conceptual level**

The conceptual level is concerned with concept(abstract), an idea of what something is or how it works; something formed in the mind; a mental image.

There exist different models used for the conceptual level to represent all the data elements likely to belong to the database but the most used is the Entity- Relationship Model (ERM) It uses the main concepts like entities, attributes and relationships.

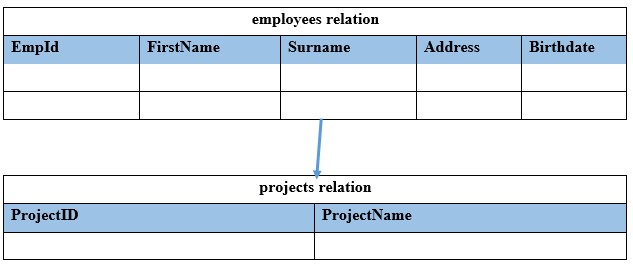
For example, when an employee works on many projects, a relationship exists between the employee and each project.

If an employee is identified by EmpId, FirstName, Surname, Address and Birthdate while a project is identified by its ProjectId and ProjectName, the diagram representing the Entity Relationship is the following.



**5.1.2 The logical level**

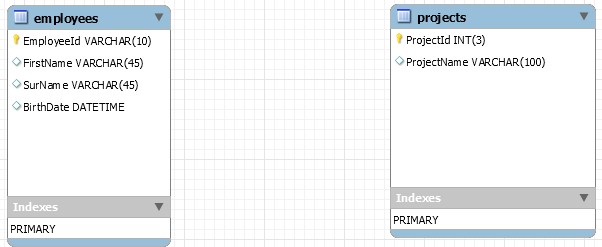
The logical level makes it possible to create relational structures enabling us to put into practice the conceptualization by imagining a relational Database Management System (DBMS). It is characterized by clear and sound reasoning. At this level there exist different models like network data models and hierarchical data models and relational model. Currently the mostly used is the Relational model. In the case of database having Employee and Project entities the relational model is represented by the following tables.



**5.1.3 Physical level**

The **physical level** is concerned with how data will be encoded and stored. It consists of the practical application (Database Management Systems - DBMS) of all the preceding theories by using computers together with its software to create a database. It deals with storage and processing performance, volumetric (time & space), partitioning and distribution.

The most known development tools to use are MS Access, MySQL (My Structured Query Language), SQL (Structured Query Language) and NoSQL.



## **5.2. DATABASE DESIGN STEPS**

The most important thing to do to start designing a database is to think ahead.

When a case that needs a database creation is presented, before to switch on the computer, it is better to think about the type of information to work with and the types of questions that the database should answer, what information needs to be stored and what specifically are the links between them. The next phases are to verify all requirements specifications, to represent the data with diagram, and to plan the database.

A well-designed database performs well and adapts to future needs by giving users access to essential information. Poor planning often results in a database that fails to meet overlooked needs.

In planning the database, regardless of its size and complexity the following basic steps are used:

1. Investigate the information.
2. Identify the objects.
3. Model the objects.
4. Identify the types of information for each object.
5. Identify the relationships between objects.
6. Database optimization through normalization.
7. Data entry and manipulation

**5.2.1. Investigate information.**

Before creating a database, there is a need of good understanding of the problem that the database is expected to solve. If the database is to replace the traditional method, file-based approach method, then the existing system will give most of the information needed.

During investing information, there is a need to work with everyone involved in the existing system to see what is needed from the new database. Gathering techniques include collect copies of customer information, management reports, and any other documents that are part of the existing system, because these will be useful in designing the database and the interfaces.

**5.2.2. Identifying the important entities and their attributes**

During the process of gathering/investigating information, the key objects or entities that will be managed by the database must be identified. The object can be a tangible thing, such as a person

(for example student, employee, and patient) or a product, or it can be a more intangible item, such as a *department* in an institution, a *Combination* in a school. Each distinct item in the database should have a corresponding table for which column titles are attributes of the entity

**5.2.3. Identifying the Relationship Between entities**

One of the strengths of an E-R database is the ability to relate or associate information about various items in the database.

Isolated types of information can be stored separately, but the database can combine data when it is required. Identifying the relationship between entities in the design process requires looking at the entities, determining how they are logically related, and adding relational columns that establish a link from one table to another.

**5.2.4. Modeling the objects**

As the objects in the system and their attributes are identified, they are recorded in a way that represents the system visually.

To record them the database model including *Relational model* and *Entity R*elational *model*

**5.2.5. Identifying the types of information for each object**

After identifying the primary objects/entities in the database as candidates for tables, the next step is to identify the types of information that must be stored for each object.

These are the columns in the table of the object.

Fields/columns should be kept simple, the more atomic your fields the more flexible will be your database.

For example, in a database of **names and addresses**, you would keep each part of the person's name as a separate field.

**5.2.6. Database optimization through normalization**

One of the most important steps to consider when designing a database is database definition. If tables are not set up properly, it can cause a lot of headaches down the road the time of extracting/retrieving required data. Understanding the rules of normalization enforces redundancy elimination and inconsistent dependency in database designs.

**5.2.7. Data entry and manipulation**

The goal of data entry is to create data that are valid and well organized to assure their quality during extraction. Well stored data leads to data consistency.

## **5.3. RELATIONAL MODEL**

**5.3.1. Introduction**

The relational data model was introduced by C. F. Codd in 1970. Currently, it is the most widely used data model.

The relational data model describes the world as “a collection of inter-related relations (or tables).” **Relation**

A relation, also known as a table or file, is a subset of the Cartesian product of a list of domains characterized by a name. You can also think of it this way: an attribute is used to define the record and a record contains a set of attributes.

The following are the key component to know when we are talking about relation: Relation, Tuple, Attribute, Cardinality, Degree, Primary key, Domain.

**Equivalent Database Concepts**

* Relation <=> Table
* Tuple <=> Row or record
* Attribute <=> Column or field
* Cardinality <=> Number of rows
* Degree <=> Number of columns
* Primary key <=> Unique identifier
* Domain <=> Pool of legal values

**Table**

A **table** is a collection of data elements organized in terms of rows and columns. Each Row is known as record and the data items are known as fields. Tables contain data about one type of item, person or event, for example:

* + a table of patients
  + a table of a student
  + a table of teacher
  + a table of books
  + a table of doctor’s appointment

For a book the fields could include:

* + title
  + author
  + ISBN
  + Publication house

**Column**

A database stores pieces of information or facts in an organized way. The principal storage units are called columns or fields or attributes. These house the basic components of data into which your content can be broken down.

**Records**

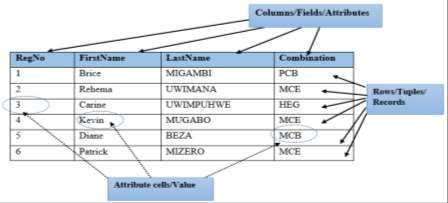
Each row represents a group of related data values, such as a customer or an employee. A row, or record, is also known as a tuple

**Cell**

The link/intersection between column and record is known as cell.

It also needs to be available so that they can be reconstituted into their whole form, the basis of all databases.

A simple table below gives us the clearest picture of how records and fields work together in a database storage project.



**Null value**

In many situations every row and column will contain data, but there are cases where it makes sense for some columns to not contain a value.

In our example RegNo cannot be null because it is unique but Combination can be null because it is optional field.

**Degree**

The degree is the number of attributes in a table. In our example above the degree is

**4**.

**Domain**

A domain is the original sets of atomic values used to model data. A domain is a set of acceptable values that a column is allowed to contain.

**For example:**

* The domain of Marital Status has a set of possibilities: Married, Single, Divorced, Widowed.
* The domain of Shift has the set of all possible days: {Monday, Tuesday, Wednesday, Thursday, Friday, Saturday and Sunday}.
* The domain of Salary is the set of all floating-point numbers greater than 0 and less than 200,000

FRW.

* The domain of First Name and Surname is the set of character strings that represents names of people.

**Datatype**

Each field in a record has its own data type. The data types in the fields can be text, alphanumeric, numeric, and Boolean or date/time.

**5.3.2 Queries in design view**

Data in a Microsoft Access database is stored in various interlinked tables. A database also has forms for data entry though you could enter data directly in tables and queries for manipulating your data and permitting information to be retrieved from a table. It also allows filtering so only the required records and fields are seen.

**Steps for creating database in Microsoft Access**

* Open Microsoft Access
* Click on Blank database in the task pane
* Name your database file (say **Company**) and click the Create button to create the database.

**Steps for creating a table in Microsoft Access-Design View**

1. Open a created database in access DBMS
2. Click on create tab from menu bar
3. In table group click on table design icon
4. Fill field names, data type and their descriptions

Data in an Access table is stored in various fields. Those fields have different properties such as fieldname, size and data types.

**Steps for creating relationship between created tables**

1. Open a created database in access DBMS
2. Click on **database tools** tab from menu bar
3. In relationships group click on **relationships** icon then the relationships panel will be displayed
4. Select a table name and click on **add** button
5. Repeat step 4 to all table you want to use
6. Click on **design** tab from menu bar
7. In tools panel click on **edit relationship** icon
8. On opened window click on **create new** button
9. Select the table names and their columns that you want to join
10. Click on **ok** button
11. Repeat 8, 9, and 10 to all pair table to be joined

After creating a table, enter the data. Note that there are various methods to create tables in Access, such as by using Table Wizard, Datasheet view or by importing tables, but this Unit restricts only to Design view.

**Steps for creating a query using design view-Query by example**

Creation of a query in design view has three rows, one to sort the fields, the other to specify whether or not to display a field, and the last to specify some criteria to select the records.

1. Click on **Create** tab of menu from bar
2. In queries grout click on **query Design** icon
3. Select tables to use then click on **Add** button
4. In Query Type group select a query commend to be used, Example **Select**
5. In query panel select the table names and field names you want to select in different tables
6. Check in show row ender the select item if you want to display them on output at run time
7. Rename the created **query**
8. In **Result** group of menu bar click on **Run** icon

After the above steps you will get a table containing the selected items as shown in the examples below:

**Query 1:** Company leader wants to display onlyEmpId, Fullname (sorted is ascending order) and salary for all employees.

* Select EmpId, EmpId, FullName, Salary and check them
* Select Sort by Ascending on FullName**.**

**Query2:** The CEO wants to see only EmpId and FullNames where Salary is less or equal to 240000.

* Select EmpId, EmpId ,FullName and check them
* Set criteria and uncheck Salary field.

**5.3.3 Dynamic queries using parameter**

A parameter is a piece of information you supply to a query right as you run it.

**Create a parameter query**

Creating a parameter is similar to adding a normal criterion to a query:

* 1. Create a **select** query, and then open the **query Design view**.
  2. In the Criteria row of the field you want to apply a parameter to, enter the text that you want to display in the parameter box, enclosed in square brackets. For example, [Enter the Doj:]
  3. Repeat step 2 for each field you want to add parameters to.

When you run the query, the prompt appears without the square brackets.

Fill in the value you’re looking for, and then click **OK**.

**Specify parameter data types**

To specify the data type for parameters in a query:

* 1. With the query open in Design view, on the Design tab, in the Show/Hide group, click Parameters.
  2. In the Query Parameters box, in the Parameter column, enter the prompt for each parameter you want to specify a data type for.

Make sure that each parameter matches the prompt that you used in the Criteria row of the query design grid.

* 1. In the Data Type column, select the data type for each parameter.

**5.4. ENTITY-RELATIONSHIP MODEL**

**5.4.1. Introduction**

The entity relationship (ER) data model has existed for over 35 years. It is well suited to data modelling for use with databases because it is fairly abstract and is easy to discuss and explain. ER models are readily translated to relations. ER models, also called an ER schema, are represented by ER diagrams.

ER modelling is based on two concepts:

* Entities, defined as tables that hold specific information (data)
* Relationships, defined as the associations or interactions between entities

**5.4.2. Entity Relationship Diagram**

The ER diagram is used to represent the conceptual database schema. In ER diagram:

Entity, Attributes and Relationships form the components of ER Diagram and there are defined symbols and shapes to represent each one of them. **Entity**

A rectangle represents an entity set.



A diamond represents a relationship.



**Entity Types**

* Entity types ->boxes
* Weak entity type -> double box



EMPLOYEE



Entity



DEPENDENT



Weak Entity

Entity types are similar to classes; they describe potential objects (entities) that will appear in the database

Weak entity types describe dependent entities, entities that depend on other entities for identity.

**Attribute**

An ellipse represents an attribute (Property).



**Link between attribute and entity set**

Lines represent linking of attributes (properties) to entity sets.



**5.4.4. Entities and entity sets**

An entity set is a set of entities of the same type that share the same properties. A noun is used to represent an entity set.

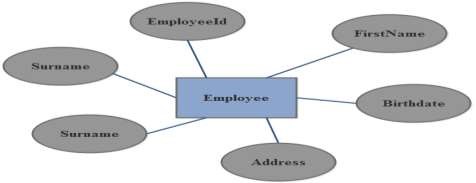
An entity is an instance of an entity set.

For example, an entity can be:

1. Concrete (TEACHER or STUDENT)
2. Insubstantial (GRADE)
3. An occurrence (EXAM)

**5.4.5. Attributes**

A characteristic of an entity, for example First name, Last name and Age. An attribute is a data item that describes a property of an entity set. Attributes determine, explain, or categorize an entity set. Attributes have values, which can be of different data types such as numbers, character strings, dates, images, sounds, and so on. In a physical model, an attribute is a named column in a table. Each table has a list of attributes (or columns).



The types of attributes are:

**Simple (atomic) attribute** – This type of attribute has a single component. It is called single valued attribute. For example, the Gender attribute has a single component with two values.

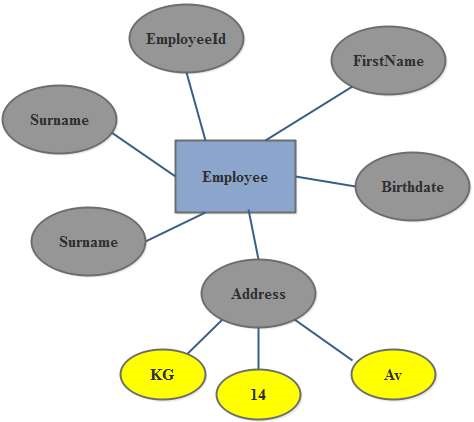
In the COMPANY database, an example of this would be: Name = {John}; Age = {23}

**Composite attribute** – A composite attribute consists of many components.

For example, the Name attribute has the components Last name and First name. So, this would be written as → Name = {KAGABO+Peter}

Address may consist of Province and District, Number, Avenue. So this would be written as →

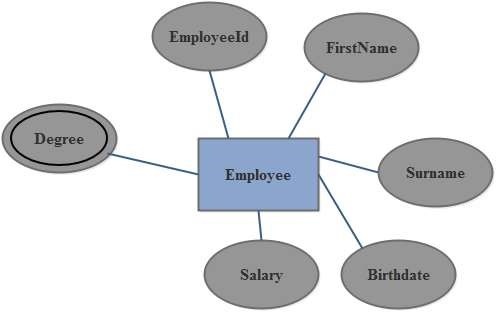
Address = {KG +‘14’ + ‘Av’}



**Single valued attribute** – This type of attribute has one value for one entity. For example, the Title attribute has a single value for each teacher.

**Multi-valued attribute** – A multi-valued attribute has many values for one entity.

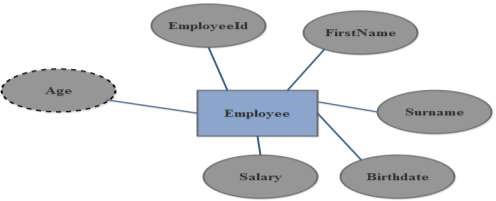
An example of a multivalued attribute from the COMPANY database, as seen in Figure below, are the degrees of an employee: BSc, MSc., PhD.



**Derived attribute** – A derived attribute has its value computed from another attribute or attributes.

A derived attribute is not a part of a table from a database, but is shown for clarity or included for design purposes even though it adds no semantic information; it also provides clues for application programmers.

An example of this can be seen in Figure below. Age can be derived from the attribute Birthdate In this situation, Birthdate is called a stored attribute, which is physically saved to the database.



**Unstable attributes** - This type of attribute have values that always change. For example, the salary of employee.

**Mandatory attributes** - Mandatory attributes must have a value. For example, in most businesses that track personal information, Name is required.

**Optional attributes** - Optional attributes may have a value or be left null; For example, address of employee

**Unique identifier** - This type of attribute distinguishes one entity from another.

For example, in a company, you can distinguish between one employee and another using a EmployeeID.

**5.4.6. Keys**

An important constraint on an entity is the key. The key is an attribute or a group of attributes whose values can be used to uniquely identify an individual entity in an entity set

In the case of **Logical Model,** you use a special approach for unique identifier. The equivalent concept for unique identifier within Logical Model is a key.

A key is a field or a set of fields that has/have a unique value for each record in the relation. You need a key to ensure that you do not meet redundancies within a relation There are several types of keys each having slightly different characteristics:

**Super key-**is defined as a set of attributes within a table that can uniquely identify each record within a table. Super Key is a superset of Candidate key.

**Candidate key** – A candidate key is an attribute or set of attributes that uniquely identifies a record in a relation.

A candidate key is unique and minimal.

It is unique because no two rows in a table may have the same value at any time. It is minimal because every column is necessary in order to attain uniqueness.

From our Company database example, if the entity is Employee (SerialNo,EmpId, First Name, Surname, Address, Contact, Birthdate, Salary, Depid), possible candidate keys are:

* ***EmpId,***

***•SerialNo***

* ***FirstName and Surname*** – assuming there is no one else in the company with the same full name
* ***Surname and Depid*** – assuming two people with the same Surname don’t work in the same Department

**Composite keys** – these keys have multiple attributes.

Using the example from the candidate key section, possible composite keys are:

* ***First Name and Surname***
* ***Surname and Depid***

**Primary key** – A primary key is one of the candidate keys from a relation. Every relation must have a primary key. A primary key shall be at least: - ***Stable.*** The value of a primary key must not change or become null throughout the life of the entity.

The primary key is indicated in the ER model by underlining the attribute.

For example, consider an employee record; using the Age field as the primary key would not be appropriate because the value of the primary key must not change over time.

It should be Minimal. The primary key should be composed of the minimum number of fields to ensure the occurrences are unique.

In this case Employee (**EmpId**, First Name, Surname, Address, Contact, Birthdate, Salary, Depid), **EmpId** is primary key

**Secondary key**

A secondary key is an attribute used strictly for retrieval purposes (can be composite), for example:

Phone and Surname.

**Alternate key** – An alternate key is any candidate key that is not chosen to be the primary key. It may become the primary key if the selected primary key is not appropriate.

**Simple keys** – these keys have a single attribute.

**Foreign keys** – these keys exist usually when there are two or more relations. An attribute from one relation has to exist in the other(s) relation. A foreign key is a field (or fields) that points to the primary key of another table. The purpose of the foreign key is to ensure referential integrity of the data. In other words, only values that are supposed to appear in the database are permitted

In the Company database example below, Depid is the foreign key

(**EmpId**, First Name, Surname, Address, Contact, Birthdate, Salary, Depid)

**Attributes and Keys**

* Key attributes must be unique for each entity
* Keys are used to identify particular entities
* Partial keys are only partially unique used for weak entity types



Age



Attribute



Key Attribute



SSN



Partial Key Attribute



Date

**Entity Types and Attributes**

All regular entity types must have a key attribute or set of key attributes, Weak entity types must have partial keys, Weak entities get part of their key (and part of their identity) from some related entity.



EMPLOYEE



Pho



EID



Na

me



DEPENDENT



Age



Name

**Nulls**

A null is a special symbol, independent of data type, which means either unknown or inapplicable.

It does not mean zero or blank.

Features of null include:

* No data entry
* Not permitted in the primary key
* Should be avoided in other attributes
* It can represent

1. an unknown attribute value
2. a known, but missing, attribute value
3. a “not applicable” condition

**Example of null**

Employee Salary table

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| **EmpId** | **JobTitle** | **Department** | **Salary** | **Commission** |

1. Sales Manager Sales 400,000 43000
2. Logistic Officer Sales 350,000 null
3. Director General Administration 1,000,000 null

In this case commission can left null

**Relationship sets**

A relationship set is a set of relationships between two or more sets of entities, and are regularly represented using a verb.

A relationship is an instance of a relationship set and establishes an association between entities that are related. These relationships represent something important in the model. Example: an employee works on a project.

A relationship set always exists between two entity sets (or one entity set relating to itself). You need to read a relationship set in double sense, from one entity set to the other.

**Relationships representation**

Relationships => diamonds

Identifying relationship => double diamond



WorksOn



Relationship



Identifying Relationship



DependentOf

Relationships indicate a meaningful connection between two entity types, Relationships may have attributes, but they cannot have key attributes. Identifying relationships connect a weak entity type to some other entity type indicates where the weak entity gets a key to complete its own partial key

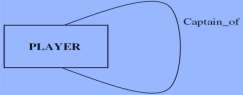
**5.4.7. Degree of a relationship**

The degree of a relationship refers to the number of associated entities. The degree of a relationship can broadly be classified into *unary, binary, and ternary relationship*.

**5.4.7.1. Unary Relationship**

The unary relationship is known as recursive relationship.

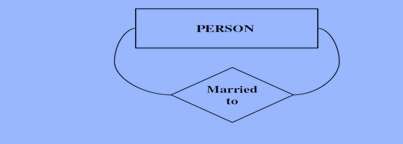
In the unary relationship the number of associated entities is one. An entity is related to itself is known as **recursive relation**.



**Roles and Recursive Relations**

When an entity set appears in more than one relationship, it is useful to add labels to the connecting lines. These labels are called roles.

In this example, labels “Husband” and “wife” are referred to as roles.



**Wife**

**Husband**

**5.4.7.2 Binary Relationship**

In a binary relationship, two entities are involved. Consider the example where each staff will be assigned to a particular department. Here the two entities are STAFF and DEPARTMENT.



**5.4.7.3. Ternary Relationship**



In a ternary relationship, three entities are simultaneously involved. Ternary relationships are required when binary relationships are not sufficient to accurately describe the semantics of an association among three entities.

**Example**

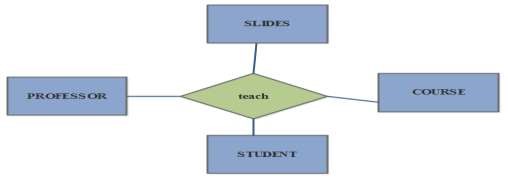
Consider the example of employee assigned a project.

Here we are considering three entities EMPLOYEE, PROJECT, and LOCATION. The relationship is “assigned-to.” Many employees will be assigned to one project hence it is an example of one-to-many relationship.

**5.4.7.4. Quaternary Relationships**

Quaternary relationships involve four entities. The example of quaternary relationship is “A professor teaches a course to students using slides.” Here the four entities are PROFESSOR, SLIDES, COURSE, and STUDENT.

The relationships between the entities are “Teaches.”



The degree of relationship, which determines how many instances of an entity relate to a single instance of another entity is called Cardinality.

**5.4.8. Classifications or types of Relationships**

Based on **cardinality**, the different types of relationships that can exist between entities are:

**5.4.8.1. One-to one (1:1)**

A single instance of an entity can relate to only one instance of the other entity. It is the relationship of one entity to only one other entity, and vice versa. It should be rare in any relational database design. In fact, it could indicate that two entities actually belong in the same table. An example from our Company database is one employee is associated with one spouse, and one spouse is associated with one employee.

**Other Examples:**

1. A person can have only one passport.
2. Another example of one-to-one relationship is House to Location. A house is obviously in only one location.

**5.4.8.2. One-to-many (1: M)**

An instance of one entity can relate to multiple instance of another instance.

The relationship that associates one record of entity A to more than one record of entity B is called one-to-many relationship.

Example of one-to-many relationship:

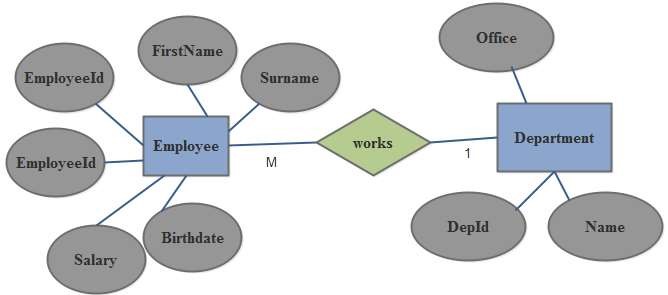
* A country having states.
* Another example of one-to-many relationship is parent–child relationship. For one parent there can be more than one child. Hence it is an example of one-to-many relationship.
* A class has many students.

**5.4.8.3. Many-to-One Relationship Type (M: 1)**

The relationship between EMPLOYEE and DEPARTMENT is an example of many-to-one relationship.

There may be many EMPLOYEES working in one DEPARTMENT.

Hence relationship between EMPLOYEE and DEPARTMENT is many-to-one relationship

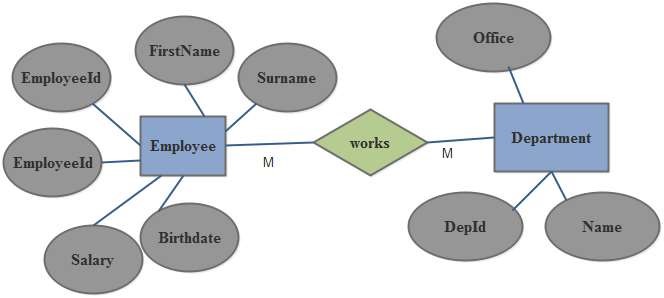


**5.4.8.4. Many-to-many (M: M):**

Multiple instances of an entity can relate to multiple instances of another entity. For our company database. The relationship between employee and department is many-to-many relationship.

Other examples are:

* A customer can purchase more than one book.
* The relationship between TEACHER entity and STUDENT entity is an example of many-to-many relationship. One teacher teaches many students and one student is taught by many teachers.
* Employee and project, many employees can work on many projects



**Participation and Cardinality**

Participation and cardinality define constraints on relationships; participation indicates whether an entity is required to take part in a relationship, cardinality ratios and structural constraints place limits on the number of entities that may participate in a relationship.

**Participation Constraints**

* Total participation  double or thick line indicates required participation
* Partial participation  thin line indicates optional participation



WorksFor



Total Participation



EMPLOYEE



DEPARTMENT



WorksOn



Partial Participation



EMPLOYEE



PROJECT

## **Relational Integrity Constraints**

Every relation in a relational database model should abide by or follow a few constraints to be a valid relation, these constraints are called as **Relational Integrity Constraints**.

The three main Integrity Constraints are:

1. Entity integrity Constraints
2. Domain Constraints
3. Referential integrity Constraints

***Entity integrity*** concerns the concept of a[**primary key**](http://en.wikipedia.org/wiki/Primary_key). Entity integrity is an integrity rule which states that every table must have a primary key and that the column or columns chosen to be the primary key should be unique and not null.

***Referential integrity*** concerns the concept of a [**foreign key**](http://en.wikipedia.org/wiki/Foreign_key). The referential integrity rule states that any foreign key value can only be in one of two states. The usual state of affairs is that the foreign key value refers to a primary key value of some table in the database.

***Domain integrity*** specifies that all columns in relational database must be declared up on a defined domain. The primary unit of data in the relational data model is the data item. Such data items are said to be non-decomposable or atomic. A domain is a set of values of the same type. Domains are therefore pools of values from which actual values appearing in the columns of a table are drawn.

## **5.5. Database optimization through normalization**

**5.5.1. Introduction**

Another method for designing a relational database is to use a process commonly known as normalization. The goal is to generate a set of relation schemas that allows us to store information without unnecessary redundancy, yet also allows us to retrieve information easily. The approach is to design schemas that are in an appropriate normal form. Therefore, normalization is a part of the database design process.

**Normalization** is the process of minimizing **redundancy** from a relation or set of relations.

Redundancy in relation may cause insertion, deletion and update on anomalies. So, it helps

to minimize the redundancy in relations. **Normal forms** are used to eliminate or reduce

redundancy in database tables.

**5.5.2. Normal Forms**

The most common normal forms are First Normal Form (1NF), Second Normal Form (2NF), Third Normal Form(3NF) and Fourth Normal Form(4NF).

**First Normal Form**

The conversion to first normal form (lNF) requires splitting the data into two groups. Tables without repeating group are said to 1NF.

A relation is in first normal form if and only if the domain of each attribute contains only atomic (indivisible) values, and the value of each attribute contains only a single value from that domain

**Example:**

A database has been designed to store data about sellers and the products they have sold.

The following facts help to define the structure of the database:

* each seller works in a particular shop
* each seller has a unique Surname
* each shop has one or more sellers
* each product which is sold is manufactured by one company only
* each seller can sell any of the products
* the number of products that each seller has sold is recorded

**Table: SHOPSALES**

|  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- |
| **SurName** | | **ShopName** | **ProductName** | **NumberofProducts** | **Manufacturer** | |
| KAGABO | CompuTech Ltd | Digital Camera  Printer  Laptop | 3  2  6 | Nikon  Canon  HP |
| KALISA | DigiTech | Hair dryer  Electric shaver | 1  8 | Panasonic  Phillips |
| KAMALI | CompuTech Ltd | Digital Camera  Printer  Modem | 2  8  4 | Nikon  Canon  Huawei |

Mobile Phone 3 Samsung

This table is not in 1NF because:

1. SHOPSALES table has repeated group of attributes

(ProductName, Manufacturer)

(ProductName, NumberofProducts),

(NumberofProducts, Manufacturer)

1. each seller has a number of products
2. SurName and ShopName would need to be repeated for each record

The following table is in 1NF because it does not contain repeating groups

**Table: SHOPSALES**

|  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- |
| **SurName** | | **ShopName** | **ProductName** | **NumberofProducts** | **Manufacturer** | |
| KAGABO | CompuTech Ltd | Digital Camera | 3 | Nikon |
| KAGABO | CompuTech Ltd | Printer | 2 | Canon |
| KAGABO | CompuTech Ltd | Laptop | 6 | HP |
| KALISA | DigiTech | Hair dryer | 1 | Panasonic |
| KALISA | DigiTech | Electric shaver | 8 | Phillips |
| KAMALI | CompuTech Ltd | Digital Camera | 2 | Nikon |
| KAMALI | CompuTech Ltd | Printer | 8 | Canon |
| KAMALI | CompuTech Ltd | Modem | 4 | Huawei |
| KAMALI | CompuTech Ltd | Mobile Phone | 3 | Samsung |

**Second Normal Form (2NF)**

For the second normal form, the relation must first be in 1NF. The relation is automatically in 2NF if, and only if, the Primary Key comprises a single attribute.

To move to 2NF, a table must first be in 1NF.

The database is changed to the following design:

SELLER (SurName, ShopName)

PRODUCTSSOLD (SurName, ProductName, NumberofProducts, Manufacturer)

**Table: SELLERS**

|  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- |
| **SurName** | | | **ShopName** | | | |
| KAGABO | | | CompuTech Ltd |  | |
| KALISA | | | DigiTech |  | |
| KAMALI  **Table: PRODUCTS SOLD** | | | CompuTech Ltd |  | |
| **SurName ProductName** | | | **NumberofProducts** | **Manufacturer** | |
| KAGABO | Digital Camera | 3 | Nikon |
| KAGABO | Printer | 2 | Canon |
| KAGABO | Laptop | 6 | HP |
| KALISA | Hair dryer | 1 | Panasonic |
| KALISA | Electric shaver | 8 | Phillips |
| KAMALI | Digital Camera | 2 | Nikon |
| KAMALI | Printer | 8 | Canon |
| KAMALI | Modem | 4 | Huawei |
| KAMALI | Mobile Phone | 3 | Samsung |

Table **SELLER** is in Second Normal Form because attribute ShopName depends on primary key SurName; the same on **PRODUCTSSOLD** where NumberofProducts and Manufacture attributes depends on composite primary key SurName, ProductName

The link between these two tables is: primary key of SELLER table is FirstName, links to

FirstName in PRODUCTSSOLD table, FirstName in SalesProducts table is foreign key

**Third Normal Form (3NF)**

To be in third normal form, the relation must be in second normal form. Also all transitive dependencies must be removed; a non-key attribute may not be functionally dependent on another non-key attribute.

In our case on table **PRODUCTSSOLD**, Manufacturer attribute is dependent on ProductName, which is not the primary key of the **PRODUCTSSOLD** table therefore there is no key dependency. **Process for 3NF:**

1. Eliminate all dependent attributes in transitive relationship(s) from each of the tables that have transitive relationship.
2. Create new table(s) with removed dependency.
3. Check new table(s) and modified table(s) to make sure that each table does not contain contains inappropriate dependencies.

See the three new tables below.

SELLERS (SurName, Shop)

PRODUCTSSOLD (SurName, ProductName, NumberofProducts)

PRODUCTS (ProductName, Manufacturer)

**Table: SELLERS**

|  |  |  |
| --- | --- | --- |
| **SurName** |  | **ShopName** |

KAGABO CompuTech Ltd

KALISA DigiTech

KAMALI CompuTech Ltd

**Table: PRODUCTSSOLDM1**

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| **SurName** | | **ProductName** | **Manufacturer** | |
| KAGABO | Digital Camera | Nikon |
| KAGABO | Printer | Canon |
| KAGABO | Laptop | HP |
| KALISA | Hair dryer | Panasonic |
| KALISA | Electric shaver | Phillips |
| KAMALI | Digital Camera | Nikon |
| KAMALI | Printer | Canon |
| KAMALI | Modem | Huawei |
| KAMALI | Mobile Phone | Samsung |

|  |  |  |  |
| --- | --- | --- | --- |
| **ProductName** | | **Manufacturer** | |
| Digital Camera | Nikon |
| Printer | Canon |
| Laptop | HP |
| Hair dryer | Panasonic |
| Electric shaver | Phillips |

Modem Huawei

Mobile Phone Samsung

**Fourth Normal Form (4NF)**

Fourth normal form eliminates independent many to one relationship between columns.

A relation is in Fourth Normal Form:

* a relation must first be in Boyce-Codd Normal Form.
* given relation may not contain more than one multivalued attribute.

The multi valued dependency X→Y holds in a relation R if whenever there is two tuples of R that same in all the attributes of X, then we can swap their Y components and get two new tuples that are also in R.

**Table: STUDENTS**

|  |  |  |  |
| --- | --- | --- | --- |
| **StudentId** | **Subject** | **Clubs** | |
| 110 | Mathematics | Anti SIDA |
| 110 | Computer Science | Anti SIDA |
| 110 | Mathematics | Anti drugs |
| 110 | Computer Science | Anti drugs |
| 151 | Music | Environmental |

Primary key→{StudentId , Subject , Clubs}

* Many StudentId have same Subject.
* Many StudentId have same Clubs.

Thus, violates 4NF.

To convert to 4NF, the table **STUDENTCLUB** is changed to the following design:

**STUDENTSUBJECT** {StudentId, Subject}

|  |  |
| --- | --- |
| **StudentId** | **Subject** |

110 Mathematics

110 Computer Science

151 Music

**STUDENTCLUBS** {StudentId , Clubs}

|  |  |
| --- | --- |
| **StudentId** | **Clubs** |

110 Anti SIDA

110 Anti Drugs

151 Environmental

## 

## **UNIT 6: POINTERS AND STRUCTURE IN C++**

**6.1 Pointers in C++**

**6.1.1 Definition of Pointers**

A **pointer** is a variable which contains the address in memory of another variable. There can be a **pointer** to any variable type. The unary or monadic operator & gives the “address of a variable”. The indirection or dereference operator \* gives the “contents of an object pointed to by a **pointer**”.

C++ gives the power to manipulate the data in the computer's memory directly. The space in the memory can be assigned and de-assigned. This is done using pointer variables. Pointer variables are variables that points to a specific address in the memory pointed to by another variable.

**6.1.2 Dereference operators**

The asterisk sign (\*) in the cout statements is called the dereference operator. If the dereference operator is used, the “value pointed by” a pointer is got. The statement:

cout<< \*ptr; means to print the content of the memory space pointed by ptr.

**Example: use a pointer in calculating area of circle**

*#include<iostream> using namespace std;*

*int main ()*

*{*

*float PI = 3.14; float \*PIptr; float r=5, A; floatx;*

*PIptr=&PI; //PIptr contains the address of PI x=\*PIptr; // value stored in PI(3.14)is assigned to x.*

*A=r\*r\*x;*

*cout<<"The area of circle is"<<A;*

*return 0;*

**}**

**6.1.3 Reference operator**

The ampersand sign (&) is called the reference operator. If the reference operator is used, it will get the “address of” a variable. The ptr = &x; means that it stores the address of the variable x in the pointer ptr.

Example:

*#include<iostream> using namespace std; int main ( )*

*{*

*int x;*

*int \*ptr; x = 5;*

*ptr = &x; cout<< \*ptr;*

*return 0;*

*}*

**Note:** If you forget to place **\*** (in front of the pointer) in the **cout** statement, you will print the address of integer x.

**Example of pointer demonstrating the use of & and \*:**

#include <iostream>*using namespace std;*

*int main()*

*{*

*/\* Pointer of integer type, this can hold the address of an integer type variable. \*/ int \*p; int var = 10;*

*/\* Assigning the address of variable var to the pointer \* p. The p can hold the address of var because var is \* an integer type variable. \*/*

*p = &var; cout<< "Value of variable var is:"<< var <<endl;*

*cout<<"\n Value of variable var is: "<< \*p <<endl;*

*cout<<"\n Address of variable var is: "<<&var <<endl;*

*cout<<"\n Address of variable var is: "<< p <<endl;*

*cout<<"\n Address of pointer p is: "<<&p <<endl;*

*return 0;*

*}*

**6.1.4 Declaring pointer variables**

A pointer is declared by using the \* (asterisk) operator before an identifier.It is necessary to initialize the pointer before you can use a pointer in for instance a cout statement. Due to the ability of a pointer to directly refer to the value that it points to, it becomes necessary to specify in its declaration which data type a pointer is going to point to. It is not the same thing to point to a char as to point to an int or a float.

**Pointer declaration Syntax**:

***data\_type \*pt\_name;***

Where ***data\_type*** is the data type of the value that the pointer is intended to point to.

The following are three declarations of pointers and all of them are pointers and will occupy the same amount of space in memory:

*int \* number; char \* character; float \* greatnumber; some Pointer declaration examples*

*int \*iptr; // iptr is pointer to int*

*char \*cptr; // cptr is a pointer to char*

*float \*fptr; // fptr is a pointer to float*

*List \*lptr; // lptr is a pointer to List object*

**6.1.5 Initialization of Pointer Variable**

Pointers can be initialized to point to specific locations at the very moment they are defined, you need to initialize a pointer by assigning it a valid address. *pointer\_variable = &variable;*

Example of initialization:

*int \*ptr; // pointer to int declared, value undefined int x = 5; // int declared and initialized to 5*

*cout<< x; //prints 5*

*cout<< \*ptr; //Error! Prints undefined value, since ptr not initialized ptr =&x; //ptr now contain value of x*

*cout<<\*ptr; // Prints 5*

Example program of Declaring and Initializing pointer:

*#include<iostream>*

*using namespace std;*

*int main ()*

*{*

*// Declaring Variables and Pointer int Num=10; int \*Ptr;*

*Ptr=&Num;*

*// Printing Values of Pointer*

*cout<<"\n Value Of Num :" << Num <<endl ;*

*cout<<"\n Address Of Num :" <<&Num <<endl;*

*cout<<"\n Value Of Ptr :" <<Ptr<<endl;*

*cout<<"\n Address Of Ptr :" <<&Ptr<<endl;*

*cout<<"\n Ptr's Pointer Value:" << \*Ptr<<endl;*

*cout<<"\n Ptr Equal to &Num :" << \*(&Num) <<endl;*

*// Change Values of Using Pointer*

*cout<<"\n\n We Can Change Value Of Num,Without Using Num" <<endl;*

*\*Ptr=100;*

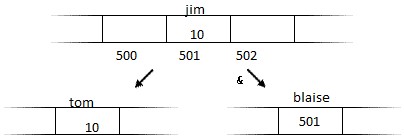
*cout<<"\n Address Of Num :"<<&Num <<endl; cout<<"\n Ptr's Pointer Value:"<<\*Ptr<<endl;*

*return 0;*

*}*

**6.1.6 Graphical representation of pointers for referencing memory allocation.** Consider the following, *int jim,tom,\*blaise; jim = 10; tom= jim; blaise = &jim;*

The following is graphical representation of the value contained in each variable after the execution of this, *are* shown in *the* following diagram:



First, we have assigned the value 10 to jim (a variable whose address in memory we have assumed to be 501).

The second statement copied to tom the content of variable jim which is 10. The third statement copies to blaise not the value contained in jim but a reference to it (its address, which we have assumed to be 501). The reason is that in this third assignment operation we have preceded the identifier jim with the reference operator (&), so we were no longer referring to the value of jim but to its reference (its address in memory).

**6.1.7Incrementing a Pointer**

Incrementing a pointer is generally used in array because we have contiguous memory in array and we know the contents of next memory location. Incrementing pointer variable depends upon data type of the Pointer variable

Three Rules should be used to increment pointer

i.Address + 1 = Address

ii.Address++ = Address iii.++Address = Address

The following program increments the variable pointer to access each succeeding element of the array.

*#include<iostream>*

*using namespace std;*

*const int MAX = 3;*

*int main ()*

*{*

*int var [MAX] = {10, 100, 200}; int \*ptr;*

*// let us have array address in pointer.*

*ptr = var;*

*for (int i =0 ;i< MAX; i++)*

*{*

*cout<< "Address of var[" <<i<< "] = "; cout<<ptr<<endl;*

*cout<< "Value of var[" <<i<< "] = "; cout<< \*ptr<<endl; // point to the next location*

*ptr++;*

*}*

*return 0;*

*}*

**6.1.8 Decrementing a Pointer**

As shown in example below, the program decrementing a pointer, which decreases its value by the number of bytes of its data type

*#include <iostream>*

*using namespace std;*

*const int MAX = 3;*

*int main ()*

*{*

*int var [MAX] = {10, 100, 200}; int \*ptr;*

*// let us have address of the last element in pointer.*

*ptr = &var[MAX-1];*

*for (int i = MAX; i> 0; i--)*

*{*

*cout<< "Address of var[" <<i<< "] = "; cout<<ptr<<endl;*

*cout<< "Value of var[" <<i<< "] = "; cout<< \*ptr<<endl; // point to the previous location*

*ptr--;*

*}*

*return 0;*

*}*

**6.1.9 Arrays and pointers**

The name of an array is a pointer to the first element in the array.

*int a[10]; int \*aptr = a;*

*a // is equivalent to &a[0]*

*aptr = a // is equivalent to aptr = &a[0];*

*aptr+5 // is equivalent to &a[5]*

*\*(aptr+5) // is equivalent to a[5]*

**Example:**

*#include <iostream>*

*using namespace std;*

*int main()*

*{*

*float arr[10]; float \*ptr;*

*cout<< "Displaying address using arrays: " <<endl; for (int i = 0; i< 10; ++i)*

*{*

*cout<< "&arr[" <<i<< "] = " <<&arr[i] <<endl;*

*}*

*// ptr = &arr[0] ptr = arr;*

*cout<<"\n Displaying address using pointers: "<<endl; for (int i = 0; i< 10; ++i)*

*{ cout<< "ptr + " <<i<< " = "<<ptr + i<<endl;*

*}*

*return 0;*

*}*

Different pointer **ptr**is used for displaying the address of array elements **arr**. Pointers are the variables that hold address. Not only can pointers store address of a single variable, it can also store address of cells of an array.

**6.2. Memory allocation**

**6.2.1 Definition**

Memory allocation means reserving memory for specific purposes. It is a process by which computer programs and services are assigned with physical or virtual memory space, the process of reserving a partial or complete portion of computer memory for the execution of programs and processes. Once the program has finished its operation, the memory is released and allocated to another program or merged within the primary memory. Programs and services are assigned with a specific memory as per their requirements when they are executed

**6.2.2 Dynamic memory allocation in C++**

Up to this level, all memories needed in programs were determined before the program execution by defining the variables where it is reserved for. But there may be cases where the memory needed in a program can only be determined during runtime.

When a memory is dynamically allocated, the operating system is asked to reserve some of that memory for that program’s use. If it can fulfill this request, it will return the address of that memory to the current application. From that point forward, the application can use this memory as it wishes. When the application is done with the memory, it can return the memory back to the operating system to be given to another program. This allows to obtain more memory when required and release it when not necessary.

**6.2.3 Static memory (or Compile Time) allocation**

The memory for your variables is allocated when the program starts, the size is fixed when the program is created

Statically allocated variables have their storage allocated and initialized before main starts running and are not deallocated until main has terminated. Statically allocated local variables are not reinitialized on every call to the function in which they are declared. A statically allocated variable thus has the occasionally useful property of maintaining its value even when none of the functions that access the variable are active.

**6.2.4 Advantages of dynamic memory allocation over static memory allocation**

An advantage is that the program determines how much memory it needs at run time and allocate exactly the right amount of storage and released when it's no longer needed.

* Dynamic Memory Allocation
  + - Memory allocated "on the fly" during run time
    - dynamically allocated space usually placed in a program segment known as the heap or the free store
    - Exact amount of space or number of items does not have to be known by the compiler in advance.

oFor dynamic memory allocation, pointers are crucial

* Static (or Compile Time) Allocation
* Memory for named variables is allocated by the compiler
* Exact size and type of storage must be known at compile time
* For standard array declarations, this is why the size has to be constant.

**6.3 C++ Structures**

**6.3.1 Introduction to C++ structure**

Structure is a collection of variables of different data types under a single name. These data elements, known as members, can have different types. To store some information about a student: Name, Age and student ID. Creation of different variables name, Age, ID needed to store this information separately.

**6.3.2 How to declare a structure in C++ programming?**

The **struct** keyword defines a structure type followed by an identifier (name of the structure).

Data structures can be declared in C++ using the following syntax:

*struct type\_name*

*{*

*member\_type1 member\_name1; member\_type2 member\_name2; member\_type3 member\_name3;*

*} object\_names;*

Where:

type\_name is a name for the structure type, object\_name can be a set of valid identifiers for objects that have the type of this structure. Within braces {}, there is a list with the data members, each one is specified with a type and a valid identifier as its name.

**6.3.3 Initializing structure in C++ programming**

Just like any variable, we can specify the initial value of a structure at the time of its definition.

Example of Initialization of structure:

*#include<iostream>*

*using namespace std; struct Book*

*{*

*char name [100]; float price;*

*int pages;*

*};*

*int main ()*

*{*

*struct Book b1= {"Computer Programing with C++",30000,250}; struct Book b2= {"Computer Networking ",15000,200}; cout<<"Book 1 Details\n"<<endl; cout<<"Title:\t"<<b1.name<<endl; cout<<"Price:\t"<<b1.price<<endl; cout<<"Pages:\t"<<b1.pages<<endl<<"\n"; cout<<"Book 2 Details\n"<<endl; cout<<"Title:\t"<<b2.name<<endl; cout<<"Price:\t"<<b2.price<<endl; cout<<"Pages:\t"<<b2.pages<<endl;*

*return 0;*

*}*

Members of object b1 and b2 are initialized. Values which will be passed to members of the structure are listed in the braces.

**6.3.4 Accessing members of a structure**

We need to specify both the structure name (name of the variable) and the member name when accessing information stored in a structure.

There are two types of operators used for accessing members of a structure:

* Member operator (.) or Dot Operator ***structure\_variable\_name.member\_name***

For Example:

We can access the member age of employee structure variable employee\_one as:

*struct employee*

*{*

*char name[50];*

*int age;*

*float salary;*

*char department[30];*

*} employee\_one = {"paul", 25, 150000.5, "ICT"};*

*int age = employee\_one.age;*

* Structure pointer operator (->) or Arrow Operator (->)

Structure pointer operator or Arrow operator is used to access members of structure using pointer variable. When we have pointer to a structure variable, then we can access member variable by using pointer variable followed by an arrow operator and then the name of the member variable.

***structure\_pointer->member\_name;***

For example:

*struct employee*

*{*

*char name [50];*

*int age;*

*float salary;*

*char department [30];*

*} employee\_one = {"Paul", 25, 150000.5, "ICT"}; struct employee \*ptr = &employee\_one;*

*int age = ptr->age;*

**//C++ Program to print the members of a structure using dot and arrow operators:**

*#include <iostream> #include <conio.h> using namespace std; struct employee*

*{*

*char name[100];*

*int age;*

*float salary;*

*char department[50];*

*};*

*int main()*

*{*

*struct employee employee\_one, \*ptr;*

*cout<<"Enter Name, Age, Salary and Department of Employee\n"; cin>>employee\_one.name>>employee\_one.age>>employee\_one.salary>>employee\_one.depar tment;*

*/\* Printing structure members using dot operator \*/*

*cout<<"Employee Details\n" <<employee\_one.name<<"\t" <<employee\_one.age<<"\t"*

*<<employee\_one.salary<<"\t" <<employee\_one.department ; /\* Printing structure members using arrow operator \*/*

*ptr=&employee\_one;*

*cout<<"\n\nEmployee Details\n"<<ptr->name<<"\t"<<ptr->age<<"\t"<<ptr-*

*>salary<<"\t"<<ptr->department; return 0;*

*}*

**6.3.6 Global use of structure**

As defined, a structure is a compound data type that contains different members of different types, **structs** are simpler to be managed by the programmer and the compiler, a better example of using a global variable, and a situation where global variables are completely necessary, is when passing a structure to a function. In that case, you must declare the structure as global so that all functions can access variables of that structure type.

**6.3.7 Pointer to structure**

Just like other variables we can use pointers to access information stored in a structure. Structures can also be pointed by pointers and store pointers. The rules are the same as for any fundamental data type. The pointer must be declared as a pointer to the structure.

**6.3.8 Nesting structure**

Structures can be nested within other structures in C++ programming. Structures can also be nested so that a valid element of a structure can also be another structure. When a structure contains another structure, it is called nested structure.

**Syntax for structure within structure or nested structure**  *struct structure1*

*{*

* *- - - - - - - - -*
* *- - - - - - - - -*

*};*

*struct structure2*

*{*

* *- - - - - - - - -*
* *- - - - - - - - - struct structure1 obj;*

*};*

**Example:**

*#include <iostream> using namespace std; struct Address*

*{*

*char SchoolName[25]; char Prov[25];*

*char District[25]; char City[25];*

*char Code[25];*

*};*

*struct Student*

*{ int Id;*

*char FName[25]; char SName[25];*

*char StudentClasse[25]; char StudentCourse[25]; float Marks;*

*struct Address Add;*

*};*

*int main()*

*{*

*int i;*

*Student S;*

*cout<< "\n\t Enter Employee Name : "; cin>>S.FName;*

*cout<< "\n\t Enter Student Second Name : "; cin>>S.SName;*

*cout<< "\n\tEnter Student Id : ";*

*cin>>S.Id;*

*cout<< "\n\tEnter Student ShoolName : "; cin>>S.Add.SchoolName;*

*cout<< "\n\tEnter Student Shoolprovince : "; cin>>S.Add.Prov;*

*cout<< "\n\tEnter Student ShoolDistrict : "; cin>>S.Add.District;*

*cout<< "\n\tEnter Student ShoolCity : ";*

*cin>>S.Add.City;*

*cout<< "\n\tEnter Student Shoolcode : ";*

*cin>>S.Add.Code;*

*cout<< "\n\tEnter Student marks : ";*

*cin>>S.Marks;*

*cout<< "\nDetails of Employees";*

*cout<< "\n\tStudent First Name: " <<S.FName; cout<< "\n\tStudent Second Name : " <<S.SName; cout<< "\n\tStudent ID : " <<S.Id;*

*cout<< "\n\tSchoolName : " <<S.Add.SchoolName; cout<< "\n\t School Province : " <<S.Add.Prov; cout<< "\n\t School District : " <<S.Add.District; cout<< "\n\t School City : " <<S.Add.City; cout<< "\n\t school code : " <<S.Add.Code; cout<< "\n\t Marks No : " <<S.Marks;*

*return 0;*

*}*

**6.3.9 Array of structure**

Arrays of structure are good for storing information of a single entity. By declaring an array of structures, you specify the number of reserved structures inside array brackets when you declare the structure variable.

**Example:**

*#include <iostream>*

*#include<string.h>*

*using namespace std; struct Employee*

*{*

*int empcode;*

*char empname[20]; char empdesig[20]; float empsalary;*

*};*

*int main ()*

*{*

*int i;*

*// Define 1 local structure variable with 20 employee information Employee emp[3];*

*for (i=1; i<= 3; i++)*

*{*

*cout<<"\nEnter info about Employee No "<<i<<":\n\n" ;*

*//get into employee information cout<<"\tEnter employee code:";*

*cin>>emp[i].empcode;*

*cout<<"\tEnter employee name:"; cin>>emp[i].empname;*

*cout<<"\tEnter employee designation:"; cin>>emp[i].empdesig;*

*cout<<"\tEnter employee salary:"; cin>>emp[i].empsalary;*

*}*

*//print the structure emp1 information to the screen cout<<"\n\nHere is the employee information:\n\n"; for(i=1;i<=3;i++)*

*{*

*cout<<"\nInformation about Employee No "<<i<<" is \n" ;cout<<"\n\t Employee code:"<<emp[i].empcode;*

*cout<<"\n\t Employee name:"<<emp[i].empname; cout<<"\n\t Employee designation:"<<emp[i].empdesig; cout<<"\n\t Employee salary:"<<emp[i].empsalary<<"\n";*

*}*

*return 0;*

*}*

## **UNIT 7: OBJECT ORIENTED PROGRAMMING IN C++**

## **7.1. Introduction to Object Oriented Programming (OOP)**

**7.1.1 Definition of Object Oriented Programming**

***O****bject-****o****riented* ***p****rogramming* (**OOP**) refers to a type of computer programming (software design) in which programmers define not only the data type of a data structure, but also the types of operations (functions) that can be applied to the data structure.

In this way, the data structure becomes an object that includes both data and functions. In addition, programmers can create relationships between one object and another.

**7.1.2 Basic Concepts of Object-Oriented Programming**

The main concepts and principlesused within Object Oriented Programming are:

* + Objects
  + Classes
  + Abstraction
  + Encapsulation
  + Inheritance
  + Polymorphism
  + Exception Handling

**Definitions of OOP concepts**

* 1. **Object**

An object is the basic unit of OOP. It is an instance of class, which has data members and use various member functions to perform tasks.

* 1. **Class**

A class is basically a blueprint of an object. It can also be defined as user defined data type but it also contains functions in it. It declares and defines what data variables the object will have and what operations can be performed on the class's object.

1. **Encapsulation**

Encapsulation means that the internal representation of an object is generally hidden from view outside of the object’s definition. Typically, only the object’s own methods can directly inspect or manipulate its fields. It can also be said data binding. Encapsulation is all about binding the data variables and functions together in class.

1. **Abstraction**

Data abstraction and encapsulation are closely tied together, because a simple definition of data abstraction is the development of classes, objects, types in terms of their interfaces and functionality, instead of their implementation details. Abstraction denotes a model, a view, or some other focused representation for an actual item.

In C++, classes can provide methods to the outside world to access & use the data variables, keeping the variables hidden from direct access, or classes can even declare everything accessible to everyone, or maybe just to the classes inheriting it. This can be done using access specifiers.

**e)Inheritance**

Inheritance as the key feature of Object-Oriented Programming is a way to reuse code of existing objects, or to establish a subtype from an existing object, or both, depending upon programming language support. In classical inheritance where objects are defined by classes, classes can inherit attributes and behavior from pre-existing classes called base classes, super classes, parent classes or ancestor classes. The resulting classes are known as derived classes, subclasses or child classes. The relationships of classes through inheritance gives rise to a hierarchy.

The derived class inherits all the features from the base class and can have additional features of its own.

1. **Polymorphism**

It is a feature, which lets programmers create functions with same name but different arguments, which will perform different actions. That means, functions with same name, but functioning in different ways. Or, it also allows users to redefine a function and provide it with a completely new definition.

1. **Exception Handling**

Exception handling is a feature of OOP, to handle unresolved exceptions or errors produced at runtime (during the running of the program).

**Advantages of Object-Oriented Programming**

One of the principal advantages of Object-Oriented Programming techniques over procedural programming techniques is that they enable programmers to create modules that do not need to be changed when a new type of object is added. A programmer can simply create a new object that inherits many of its features from existing objects. This makes object-oriented programs easier to modify.

So, Object Oriented Programming has great advantages over other programming styles.

1. **Code Reuse and Recycling**: Objects created for Object Oriented Programming can easily be reused in other programs.
2. **Data Redundancy:** Inheritance is the good feature for data redundancy. If you need a same functionality in multiple classes, you can write a common class for the same functionality and inherit that class to sub class.
3. **Data hiding:** Implementation details are hidden from other modules and other modules has a clearly defined interface.
4. **Design Benefits**: Large programs are very difficult to write. Object Oriented Programming force designers to go through an extensive planning phase, which makes for better designs with less flaws. In addition, once a program reaches a certain size, Object Oriented Programs are actually *easier* to program than non-Object-Oriented ones.
5. **Software Maintenance:** An Object-Oriented Program is much easier to modify and maintain than a non-Object-Oriented Program. So, although a lot of work is spent before the program is written, less work is needed to maintain it over time.

## **7.2 Class definition in C++**

**7.2.1 Definition of a class**

A class is defined in C++ using keyword **class** followed by the name of class.

The body of class is defined inside the curly brackets and terminated by a semicolon at the end.

**7.2.2 Syntax**

*class class\_name*

*{*

*access\_specifier\_1:*

*member1;*

*access\_specifier\_2:*

*member2;*

*...*

*};*

**Example1:**

class

*Test*

*{*

private

*:*

int

*data\_1;*

*double data\_2;*

public

*:*

void

*function\_1()*

*{*

*data\_1 =*

*2*

*;*

*}*

double

*function\_2()*

*{*

*data\_2 =*

*3.5*

*;*

return

*data\_2;*

*}*

*;*

*}*

Here, the class is named

Test

.

This class has two data members: *data\_1* and *data\_2* and two member functions: *function\_1()*and *function\_2()*.

*A Class is* an expanded concept of *data structures*, like data structures, it can contain data members, but it can also contain functions as members.

Where **class\_name** is a valid identifier for the class, object\_names is an optional list of names for objects of this class. The body of the declaration can contain *members*, which can either be data or function declarations, and optionally *access specifiers*.

Classes have the same format as plain *data structures*, except that they can also include functions and have these new things called *access specifiers (data hiding)*. An *access specifier* is one of the following three keywords: private, public or protected.

**Access specifiers** are used to identify access rights for the data and member functions of the class. There are three main types of access specifiers in C++ programming language:

* **A private** member within a class denotes that only members of the same class have accessibility. The *private* member is inaccessible from outside the class.
* **Public**members are accessible from outside the class.
* **A protected access specifier** is a stage between *private* and *public* access. If member functions defined in a class are *protected*, they cannot be accessed from outside the class but can be accessed from the derived class.

When defining access specifiers, the programmer must use the keywords: *private*, *public* or *protected* when needed, followed by a colon and then define the data and member functions under it.

|  |
| --- |
| *class AddNumbers*  *{ private: int x,y; public:*  *void sum()*  *{*  *………*  *………* |

*}*

*};*

In the code above, the member *x* and *y* are defined as private access specifiers. The member function sum is defined as a public access specifier.

General structure for defining a class is:

*class class\_name*

*{ access\_specifier:*

*data member;*

*member\_functions;*

*access\_specifier:*

*data\_member;*

*member\_functions;*

*};*

Generally, in class, all members (data) would be declared as private and the member functions would be declared as public. If no access specifiers are identified for members of a class, the members are defaulted to private access (Private is the default access level for specifiers).

|  |
| --- |
| *class AddNumbers*  *{ int a,b; public:*  *void sum()*  *{*  *………*  *………*  *}*  *};* |

In this example, for members **a** and **b** of the class**AddNumbers** there are no access specifiers identified. This means that by default **a** and **b** are private members of class **AddNumbers**.

You can have functions inside class. These functions can be either under public or private, and their syntaxes are the same. For example, if you have a function to display information of an account called *display*, and other classes should be able to access that function, the following line would be added under the public area:

*void display (accountno);*

And use the following syntax to implement the function:

*void BankAccount::display (int accountno)*

*{*

*……………;*

*………….;*

*}*

**7.2.3 Defining member functions inside or outside the class definition**

Member functions of a class can be defined either inside or outside the class definition. Inboth cases, the function body remains the same, however, the function header is different.

**7.2.3.1Inside the Class**

A member function of a class can be defined inside the class. However, when a member function is defined inside the class, the class name and the scope resolution operator are not specified in the function header. Moreover, the member functions defined inside a class definition are by default inline functions.

To understand the concept of defining a member function inside a class, consider this example. **Example**: Definition of a member function inside a class

*Class student*

*{*

*string fname;*

*float marks;*

*public:*

*void getdata();*

*//*

*declaration*

*void putdata()//definition ins*

*ide the class*

*{*

*cout<<"*

*\*

*nstudent name is: "<<fname;*

*cout<<"*

*\*

*nhis marks are: "<<marks;*

*}*

*;*

In this example, the member function putdata() is defined inside the class book. Hence, putdata()is

by default, an inline function.

Note that the functions defined outside the class can be explicitly made inline by prefixing the keyword inline before the return type of the function in the function header. For example, consider the definition of the function getdata().

*inline void student ::getdata ()*

*{*

*body of the function*

*}*

**7.2.3.2Outside the Class**

Defining a member function outside a class requires the function declaration (function prototype) to be provided inside the class definition. The member function is declared inside the class like a normal function. This declaration informs the compiler that the function is a member of the class and that it has been defined outside the class. After a member function is declared inside the class, it must be defined (outside the class) in the program.

The definition of member function outside the class differs from normal function definition, as the function name in the function header is preceded by the class name and the scope resolution operator (: :). The scope resolution operator informs the compiler what class the member belongs to.

Here is the syntax for defining a member function outside the class:

*Return\_type class\_name :: function\_name (parameter\_list)*

*{*

*// This is the body of the function*

*}*

To understand the concept of defining a member function outside a class, consider

this example.

**Example**

Let us re-write the program which converts the degree from Fahrenheit to Celsius and displays the corresponding degrees on the screen. By creating the method outside the class, we have:

|  |
| --- |
| *#include<iostream> #include<conio.h> using namespace std; class temperature*  *{ private:*  *float degreeCelcius; float tempFahr; public:*  *void conversion (float); // declaration of a method or fnct* |

*}*

*; // end of the class*

*void temperature::conversion(float a) //definition of a method*

*{*

*tempFahr=a;*

*degreeCelcius=((tempFahr*

*-*

*32)\*(0.55))*

*;*

*cout<<"After conversion the degree in Celcius is:"<<degreeCelcius<<endl;*

*// end of the function*

*}*

*int main()*

*{*

*temperature celsius; // creation of an object statically*

*float b;*

*cout<<"Enter degree in Fahrenheit:"<*

*<*

*endl;*

*cin>>b;*

*celsius.conversion (b); // calling a method*

*getch();*

*}*

Note that the member functions of the class can access all the data members and other member

functions of the same class (private, public or protected) directly by using their names. In addition, different classes can use the same function name.

## **7.3 Object in C++**

The first characteristic of an object language is to place at your disposal **the objects**.

An object can be regarded as an additional structure of information, a type of super-variable. Indeed, we know that a variable is a place in primary memory, characterized by **an address – a name - and a type** (integer, double, character, Boolean, etc).

**7.3.1 Definition:**

An *object* is an instantiation of a class. In terms of variables, a class would be the type, and an object would be the variable.

So, an object is a group of variables of various types. It thus usually gathers tens of very different information from/to each other within the same structure, thus making this information easier to handle.

By analogy with equation of Wirth, we could say that the equation of the Object-Oriented Programming (OOP) is:

Methods+Data= Objects

**7.3.2 Creating single object**

When a class is defined, only the specifications for the object is defined, the memory or storage is not allocated.

To use the data and access functions defined in the class, you need to create objects.

**7.3.2.1Syntax to define Object in C++**

classNameobjectVariableName;

You can create objects of *Test* class (defined in above example) as follows:

|  |
| --- |
| class*Test*  *{* private*:* int *data1;* float *data2;*  public*:*  void *function1() { data1 =2;}* float *function2()*  *{*  *data2 =3.5;* return *data2;*  *}*  *};*  Int*main()* |

*{*

*Test o1, o2; }*

Here, two objects ***o1*** and ***o2***of ***Test*** class are created.

In the above class *Test*, *data1* and *data2* are data members and *function1()* and *function2()* are member functions

**Example: Class and Object**

#include <iostream>

using namespace std;

class sum

{

int x, y, total;

public:

sum(int a, int b)

{

x=a;

y=b\*2;

}

void display()

{

total=x+y;

cout<<total;

}

};

int main()

{

sum s(6,5);

s.display();

return 0;

}

**7.3.2.2Creating array of objects**

|  |
| --- |
| *#include <iostream> using namespace std;*  *class MyClass {*  *int x; public:*  *void setX(int i) { x = i; }*  *int getX() { return x; }*  *};*  *int main()*  *{*  *MyClassobs[4];*  *int i;*  *for(i=0; i< 4; i++)*  *obs[i].setX(i);*  *for(i=0; i< 4; i++)*  *cout<< "obs[" <<i<< "].getX(): " <<obs[i].getX() << "\n";*  *return 0;*  *}* |

**OUTPUT**

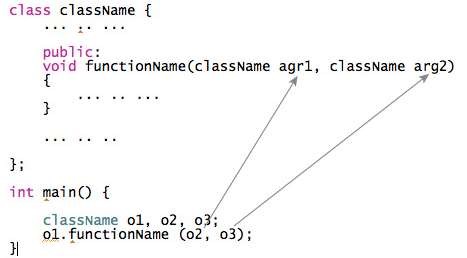
|  |  |
| --- | --- |
|  | obs[0].getX(): 0 obs[1].getX(): 1 obs[2].getX(): 2 obs[3].getX(): 3 |

**7.3.3. Passing object to function**

As it is known, we can pass (give) any type of arguments within the member function which can have any numbers of arguments. In C++ programming language, it is also possible to pass an object as an argument within the member function of class.

This is useful, when we want to initialize all data members of an object with another object, we can pass objects and assign the values of supplied object to the current object. For complex or large projects, we need to use objects as an argument or parameter.

**Syntax:**



### **7.4 Data Encapsulation and Data Abstraction in C++**

All C++ programs are composed of the following two fundamental elements

* + **Program statements (code)**: This is the part of a program that performs actions and they are called functions.
  + **Program data**: The data is the information of the program which gets affected by the program functions.

Encapsulation is an Object-Oriented Programming concept that binds together the data and functions that manipulate the data, and that keeps both safe from outside interference and exploitation. Data encapsulation led to the important OOP concept of **data hiding**.

**Data encapsulation** is a mechanism of bundling the data, and the functions that use them and **data abstraction** is a mechanism of exposing only the interfaces and hiding the implementation details from the user.

C++ supports the properties of encapsulation and data hiding through the creation of user-defined types, called **classes**. We already have studied that a class can contain **private, protected** and **public** members. By default, all items defined in a class are private. For example

|  |
| --- |
| *class Box { public:*  *double getVolume(void) {*  *return length \* breadth \* height;*  *}*  *private:*  *double length; // Length of a box double breadth; // Breadth of a box double height; // Height of a box*  *};* |

The variables length, breadth, and height are **private**. This means that they can be accessed only by other members of the Box class, and not by any other part of your program. This is one-way encapsulation is achieved.

To make parts of a class **public** (i.e., accessible to other parts of your program), you must declare them after the **public** keyword. All variables or functions defined after the public specifier are accessible by all other functions in your program.

### **7.5 Friend function in C++**

One of the important concepts of OOP is data hiding, i.e., a non-member function cannot access an object's private or protected data.

But sometimes this restriction may force programmer to write long and complex codes. So, there is mechanism built in C++ programming to access private or protected data from non-member functions.

This is done using a **friend function or/and a friend class.**

If a function is defined as a friend function then, the private and protected data of a class can be accessed using the[function.](https://www.programiz.com/cpp-programming/function)

The compiler knows a given function is a friend function by the use of the keyword **friend**. For accessing the data, the declaration of a friend function should be made inside the body of the class (can be anywhere inside class either in private or public section) starting with keyword **friend**. **Declaration of friend function in C++**

|  |
| --- |
| *class class\_name*  *{*  *... .. ...*  *friend return\_typefunction\_name(argument/s);*  *... .. ... }* |

Now, you can define the friend function as a normal function to access the data of the class.

No **friend** keyword is used in the definition.

*class className*

*{*

*... .. ...*

*friend return\_typefunctionName(argument/s);*

*... .. ...*

*}*

*return\_typefunctionName(argument/s)*

*{*

|  |
| --- |
| *... .. ...*  *// Private and protected data of className can be accessed from // this function because it is a friend function of className.*  *... .. ... }* |

**Example 1: Working of friend Function**

#include <iostream>

using namespace std;

class Number

{

private:

int a;

public:

void getNum(int x);

//declaration of friend function

friend void printNum(Number NUM);

};

//class member function definitions

void Number::getNum(int x)

{

a=x;

}

//friend function definition, no need of class name with SRO (::)

void printNum(Number NUM)

{

cout<< "Value of a (private data member of class Number): " <<NUM.a;

}

int main()

{

Number n; //Object declaration

n.getNum(1000);

printNum(n);

return 0;

}

### **7.6Polymorphism in C++**

Polymorphism is derived from two Greek words: **poly** and **morphs**. The word "poly" means many and **morphs** means forms. So, polymorphism means many forms.

**Polymorphism** is the ability of an object to take on many form.

**7.7 Overloading**

C++ allows you to specify more than one definition for a **function** name or an **operator** in the same scope, which is called **function overloading** and **operator overloading** respectively.

An overloaded declaration is a declaration that is declared with the same name as a previously declared declaration in the same scope, except that both declarations have different arguments and obviously different definition (implementation).

**Function overloading**

Function overloading is a C++ programming feature that allows us to have more than one function having same name but different parameter list.

You can have multiple definitions for the same function name in the same scope. The definition of the function must differ from each other by the types and/or the number of arguments in the argument list. You cannot overload function declarations that differ only by return type.

**Example:**

A program to find the sum of numbers using function overloading

#include <iostream>

using namespace std;

class Addition {

public:

int sum(int num1,int num2) {

return num1+num2;

}

int sum(int num1,int num2, int num3) {

return num1+num2+num3;

}

};

int main(void) {

Addition obj;

int x,y,a,b,c;

cout<<"Enter two numbers:"<<endl;

cin>>x>>y;

cout<<"Enter three numbers:"<<endl;

cin>>a>>b>>c;

cout<<"The sum of two numbers is:"<<obj.sum(x,y)<<endl;

cout<<"The sum of three numbers is:"<<obj.sum(a,b,c)<<endl;

//cout<<obj.sum(20, 15)<<endl;

//cout<<obj.sum(81, 100, 10);

return 0;

}

Output:

35

191

### **7.8 Constructors and Destructors**

**Introduction:**

Constructors and destructors determine how the objects of a class are created, initialized, copied, and destroyed. They are member functions whose names are distinguished from all other member functions because they have the same name as the class they belong to. Constructors and destructors have many of the characteristics of normal member functions.

You declare and define them within the class, or declare them within the class and define them outside--but they have some unique features.

**7.8.1. Constructor**

It's a special member function that has the same name as the class name and is automatically called

when the object of the class is created.

The main purpose of the class constructor in C++ programming is to construct an object of the class. In other word, it is used to initialize all class data members.

Notice that you are allowed to have more than one constructor. This is called overloading of functions. Therefore, if you have more than one constructor in your class you can have more than one method to declare an object of that kind.

1. **Characteristics of constructor** 
   * Name of the constructor function is same as that of the class they are part of.
   * No return type is required for the constructor function.
   * Constructor functions are automatically called at the time of object creation/declaration.
   * Constructor functions are always defined in the public type access specifier.
   * Constructor functions can be overloaded.

**b. Types of constructors**

**i. Default constructor**

Default constructor is the constructor which does not take any argument. It has noparameter.

**Syntax:**

Class\_name()

{ Constructor Definition }

**Example:**

#include <iostream>

using namespace std;

class cube

{

int side;

public:

cube()//default constructor definition

{

side=10;

cout<<"Side="<<side;

}

};

int main( )

{

cube c;

return 0;

}

In this case, as soon as the object is created the default constructor is called which initialize its data members.

A default constructor is so important for initialization of object member, that even if we do not define a constructor explicitly, the compiler will provide a default constructor implicitly.

**ii. Parameterized constructor**

It is a constructor with parameters. Using constructor, you can provide different values to data members of different objects, by passing the appropriate values as argument.

**Syntax:**

Class\_name( parameters)

{ Constructor Definition }

**Example:**

#include<iostream>

using namespace std;

class cube

{

int side;

public:

cube (int i)

{

side=i;

cout<<"Side="<<side;

}

};

int main ()

{

int a;

cout<<"Enter a number:"<<endl;

cin>>a;

cube c(a)

//cube c(10);

return 0;

}

**iii. Copy constructor**

The copy constructor is a constructor which creates an object by initializing it with an object of the same class, which has been created previously.

It's a special type of constructor which takes an object as argument, and is used to copy values of data members of one object into other object.

**Syntax:**

class class\_name

{

public:

class\_name(class\_name& obj)

{

// obj is same class another object

// Copy Constructor code

}

//... other Variables & Functions

};

Syntax: Copy Constructor Declaration In Main Function

// In Main Function

class\_name object1(params);

Method 1 - Copy Constrcutor

class\_name object2(object1);

Method 2 - Copy Constrcutor

class\_name object3 = object1;

**Example:**

#include<iostream>

using namespace std;

class Example

{

// Member Variable Declaration

int a, b;

public:

//Normal Constructor with Argument

Example(int x, int y)

{

// Assign Values In Constructor

a = x;

b = y;

cout<< "\nIm Constructor";

}

//Copy Constructor with Obj Argument

Example(Example& obj)

{

// Assign Values In Constructor

a = obj.a;

b = obj.b;

cout<< "\nIm Copy Constructor";

}

void Display() {

cout<< "\nValues :" << a << "\t" << b;

}

};

int main() {

//Normal Constructor Invoked

Example Object(10, 20);

//Copy Constructor Invoked - Method 1

Example Object2(Object);

//Copy Constructor Invoked - Method 2

Example Object3 = Object;

Object.Display();

Object2.Display();

Object3.Display();

return 0;

}

1. **Call of the constructors**

If an object has a constructor, its declaration must obligatorily comprise the corresponding arguments. For example, if the class point has a prototype constructor:

cube(int);

**The following declaration will not be correct:**

cube a; //incorrect: the constructor waits one argument cube b(3,2); // incorrect (same reason) This one on the other hand will be appropriate: cube a (7); // correct because the constructor has one argument

If there are several constructors, it is enough that the declaration comprises the arguments required by one of them. Then, if the class pointer has the following constructors: (it is the overloading of the constructors).

cube( ); //constructor 1

cube(int); //constructor 2

The following declaration will be rejected:

cube a( 5); //incorrect, no constructor has one argument

But these ones will be appropriate:

cube a; //correct: call of the constructor 1

cube b(9); //correct: call of the constructor 2

With regard to the chronology (sequence of events), we can say that:

* The constructor is called after the creation of the object;
* The destructor is called before the destruction of the object.

Up to now we noted that the call of a constructor is done during the creation of the object. But the object can be created in static or dynamic way.

**d. Calling parameterized constructors (Implicitly, Explicitly) example 1. In static: i.e. cube a (6);**

cube: name of the class

a: name of the object, created statically

(6): the parameter which is between the brackets.

**Example 2. Dynamically: i.e. cube \*pa=new cube (6);**

The constructor is called by defining a pointer to an object of desired type, then by affecting it the value returned by “new”.

**7.8.2 Overloading of constructors**

Constructors are just like other functions, they can also be overloaded, that is same named constructers can be used to give different set of outputs depending upon the kind of input given to them. The concept of constructer overloading can be easily understood by the program given below:

*# include<iostream>*

*using namespace std;*

*class work*

*{*

*int X,Y;*

*public:*

*work() //1*

*{*

*X=10;*

*Y=30;*

*}*

*work(int c) //2*

*{*

*X=c;*

*Y=2\*c;*

*}*

*work (int x,int y) //3*

*{*

*X=x;*

*Y=y;*

*}*

**7.8.3 Destructors**

C++ destructor is a special member function that is executed automatically when an object is destroyed that has been created by the constructor. C++ destructors are used to de-allocate the memory that has been allocated for the object by the constructor.

**a. Characteristics of Destructors**

A destructor has the same features as that of the constructor that are

* Name of the destructor function is same as that of the class they belong to, preceded by tilde (~)
* No return type is required for the destructor function
* Destructor functions are automatically called when the object goes out of scope
* Destructor functions are always defined in the public type access specifier

**b. Use of destructors**

When an object is created, sufficient amount of memory is allocated to it. When the scope of an object comes to an end, resource allocated to it must be freed so that it can be used elsewhere. This process can be achieved by the use of a destructor. This function is automatically invoked when the scope of an object comes to an end.

**Note:** In case the programmer does not provide for a destructor manually, the compiler provides a default destructor just as the default constructor.

The definition of the destructor is similar to that of the constructor, but its name is preceded by a tilde (“~”), and it does not have the arguments.

**Syntax:**

Its syntax is same as constructor except the fact that it is preceded by the tilde sign.

~class\_name()

{

} //syntax of destructor

**Example: use of destructor**

#include<iostream>

using namespace std;

class number

{

private:

int a,b;

public:

number(int x, int y)

{

a=x; b=y;

}

~number()

{

cout<<"The sum of two numbers is:"<<a+b<<endl;

}

};

int main()

{

int m,n;

cout<<"Enter two numbers:"<<endl;

cin>>m>>n;

number num(m,n);

return 0;

}

Destructors are less used compared to constructors. It will be used only if the objects are linked to other objects by pointers. Thus, we can use the destructor to delete the pointers towards these objects.

**c. Call of the destructor**

The destructor is called differently, according to the object to which it belongs.

* + The destructor is called automatically, at the end of the program for the static case
  + In the dynamic case, the destructor is called by the keyword “delete”, which allows freeing the memory occupied by the object.

**7.9. Inheritance in C++**

**7.9.1 Introduction**

Programmers seem to hate coding something from scratch; why code the same thing over and over again? Fortunately, it is possible to reuse code, thanks to one of the key concepts of object-oriented programming. That is inheritance, which is the subject of this chapter.

**7.9.2 Definition of Inheritance**

* + Inheritance is the process by which new classes called *derived* classes are created from existing classes called *base* classes. The derived classes have all the features of the base class and the programmer can choose to add new features specific to the newly created derived class.
  + In Object Oriented Programming, Inheritance is the process by which objects of one class acquire the properties and functionality of objects of another class. It supports the concept of hierarchical classification. For example, the bird robin is a part of the class flying bird which is again a part of the class bird.

Creating or deriving a new class using another [class](http://www.codersource.net/c/c-tutorials/cpp_tutorial_class.html)as a base is called inheritance in C++. The new class created is called a **Derived class** and the old class used as a base, is called a **Base class** in C++ inheritance terminology.

For example, a programmer can create **a *base* class named fruit** and define ***derived* classes as mango, orange, banana, etc**. Each of these derived classes, (mango, orange, banana, etc.) has all the features of the *base* class (fruit) with additional attributes or features specific to these newly created derived classes. Mango would have its own defined features, orange would have its own defined features, banana would have its own defined features, etc.

The General Form of Inheritance:

***class****derived-class: access-specifier base-class*

*{*

*...*

*... //instructions*

*...*

*};*

The two points indicate that derived-class is derived from base-class. The access-specifier is optional and if it is present can be private or public.

The mode of visibility by default is private. It indicates if the aspects of the base class are inherited in a private way or publicly.

The inheritance is also called derivation.

**7.9.3. Advantages of Inheritance**

1. **Reusability*:***

Inheritance helps the code to be reused in many situations. The base class is defined and once it is compiled, it need not be reworked. Using the concept of inheritance, the programmer can create as many derived classes from the base class as needed while adding specific features to each derived class as needed.

1. **Saves Time and Effort:**

The above concept of reusability achieved by inheritance saves the programmer time and effort; since the main code written can be reused in various situations as needed.

1. **Increases Program Structure which results in greater reliability.**

**Attention!**

Some of the exceptions to be noted in C++ inheritance are as follows.

* + The constructor and destructor of a base class are not inherited,
  + The assignment operator is not inherited,
  + The [friend](http://www.codersource.net/c/c-tutorials/cpp_tutorial_friend.html)functions and friend classes of the base class are also not inherited.

**7.9.4. Inheritance rules**

**a. class base\_class name: inheritance\_ specifier derived\_class name**

For example, if the *base* class is *student* and the derived class is peter it is specified as:

class student: public peter

The above makes peter have access to both *public* and *protected* variables of base class *student*.

Reminder about public, private and protected access specifiers:

* If a member or variables defined in a class is private, then they are accessible by members of the same class only and cannot be accessed from outside the class.
* Public members and variables are accessible from outside the class.
* If a member functions or variables defined in a class are protected, then they cannot be accessed from outside the class but can be accessed from the derived class.

**b. Base class**

A base class is a class that is created with the intention of deriving other classes from it. It is also called super/parent/old class.

**c. Child class**

A child class is a class that was derived from another, that will now be the parent class to it. It is also called sub/derived/inherited class.

The **protected** and **public** variables or members of the base class are all accessible in the derived class, but a private member variable is not accessible by a derived class.

The derived class inherits some or all the features from the parent class. A class can also inherit the properties of more than one class or more than one level.

**d. The role of the inheritance modifier and the access modifier: protected**

A protected attribute is not accessible outside the class but is accessible from the derived classes. This modifier is thus intermediate between private and public. The following table recapitulates the accesses provided by these three modifiers:

|  |  |  |
| --- | --- | --- |
| **Access**  **Modifier** | **Visibility in the children classes** | **Visibility from outside** |
| Private | No | No |
| Protected | Yes | No |
| Public | Yes | Yes |

The inheritance modifier can be public or private. It conditions the visibility of the members of the parent class in the derived class. The following table indicates to which access is associated a member with the parent class in the child class according to the inheritance modifier:

|  |  |  |
| --- | --- | --- |
| **Access in the parent class** | **Access in the child class** | |
| Public inheritance | Private inheritance |
| Private | Not accessible | Not accessible |
| protected | protected | Private |
| Public | Public | Private |

**7.9.5Types/ Forms of Inheritance**

**7.9.4.1 Single Inheritance**

In this type of inheritance one derived class inherits from only one base class. It is the simplest form of Inheritance.

**Example:**

**#include <iostream>**

**using namespace std;**

**class student**

**{**

**protected:**

**char name[30];**

**int age;**

**char gender;**

**public:**

**void getinfo()**

**{**

**cout<<"Enter student basic info:"<<endl;**

**cout<<"Name:";**

**cin>>name;**

**cout<<"\nAge:";**

**cin>>age;**

**cout<<"\nGender:";**

**cin>>gender;**

**}**

**};**

**class studentinfo:public student**

**{**

**public:**

**void display()**

**{**

**cout<<"Basic student info:"<<endl;**

**cout<<"Name:"<<name<<endl;**

**cout<<"Age:"<<age<<endl;**

**cout<<"Gender:"<<gender<<endl;**

**}**

**};**

**int main( )**

**{**

**studentinfo s;**

**s.getinfo();**

**s.display();**

**return 0;**

**}**

**7.9.4.2 Multiple inheritance**

*Multiple inheritance* is the construction in which objects inherit from more than one object type or class. This contrasts with single inheritance, where objects can only inherit from one type or class.

**Example:**

**#include<iostream>**

**using namespace std;**

**class student**

**{**

**protected:**

**int rno, m1, m2;**

**public:**

**void get() {**

**cout<< "Enter the Roll no :";**

**cin>>rno;**

**cout<< "Enter the two marks :";**

**cin>> m1>>m2;**

**}**

**};**

**class sports**

**{**

**protected:**

**int sm; // sm = Sports mark**

**public:**

**void getsm()**

**{**

**cout<< "\nEnter the sports mark :";**

**cin>>sm;**

**}**

**};**

**class statement : public student, public sports**

**{**

**int tot, avg;**

**public:**

**void display()**

**{**

**tot = (m1 + m2 + sm);**

**avg = tot / 3;**

**cout<< "\n\n\tRoll No : " <<rno<< "\n\tTotal : " << tot;**

**cout<< "\n\tAverage : " << avg;**

**}**

**};**

**int main()**

**{**

**statement obj;**

**obj.get();**

**obj.getsm();**

**obj.display();**

**return 0;**

**}**

**7.9.4.3 Multi-level inheritance**

Let us consider an example: let us assume that the result of an exam of the student is stored in the 3 different classes. Class number stores the registration number, class test stores marks obtained in the two subjects and finally the class result contains the total of the points obtained in the exam. The class result can inherit the details, such as the points obtained in the exam and the registration number of the student through the multilevel heritage.

**Example:**

**#include<iostream>**

**using namespace std;**

**class First**

**{**

**public:**

**int a, b, s;**

**void input()**

**{**

**cout<< "Enter two numbers :"<<endl;**

**cin>> a>>b;**

**}**

**};**

**class Second : public First**

**{**

**public:**

**void add() {**

**s = a + b;**

**}**

**};**

**class Third : public Second**

**{**

**public:**

**void display() {**

**cout<< "\nSumis :" << s;**

**}**

**};**

**int main() {**

**Third th;**

**th.input();**

**th.add();**

**th.display();**

**return 0;**

**}**

**7.9.4.4 Hierarchical Inheritance**

In this type of inheritance, multiple derived classes inherit from a single base class.

**7.9.4.5 Hybrid Inheritance**

There could be situations where we must apply two types or more of inheritance to develop a program.

## **UNIT8. INTRODUCTION TO VISUAL BASIC**

**8.1. Understanding Visual Basic.**

**8.1.0. General introduction**

VISUAL BASIC is a high level and **Event-driven Programming Language** which evolved from the earlier Disk Operating System (DOS) version called BASIC. (**BASIC** means: **B**eginners' **A**llpurpose**S**ymbolic **I**nstruction **C**ode).

Visual basic for DOS and Visual Basic for Windows were introduced in 1991 and evolved through the years to give Visual basic 3.0(in 1993), Visual basic 4.0 (in 1995) Visual basic 5.0 (in 1996) and now we use Visual basic 6.0 version released in 1998.

**Visual basic 6.0. has the following advantages:**

* It is easier for the user to minimize code writing.
* The user will become more familiar with visual approach for other visual languages.
* It provides Input box and Output box as interactive windows with user.
* It is very easy program language compare with others.
* The VB –IDE has been highly optimized support rapid application development

(“RAD”). It is particularly easy to develop graphical user interfaces.

* VB provides a comprehensive interactive and context-sensitive online help system.
* Visual basic is built around the .NET environment used by all Microsoft Visual language, so the is a very little that can’t done in Visual Basic that can be done in other languages (such as C#).

**Visual basic 6.0. has the following disadvantages:**

* Visual Basic is a proprietary programming language written my Microsoft, so program written in Visual Basic cannot be easily transferred to another operating system

8**.1.1. Definition of terms**

**a)Graphical User Interface.**

A graphical user interface (GUI) is an interface through which a user interacts with electronic devices such as computers, hand-held devices and other appliances. GUI representations are manipulated by a pointing device such as a mouse, trackball, stylus, or a finger on a touch screen.

Graphical user interface (GUI) is different from command line interface (CLI) or command language interpreter as command line interface (CLI) enables users to type commands in a terminal or console window to interact with an operating system.

**b)Desktop application**

A desktop application is a computer program that runs locally on a computer device, such as desktop or laptop computer, in contrast to a web application, which is delivered to a local device over the Internet through browser from a remote server. Different user environments can impact whether a desktop or a web application is the best solution for your needs.

**Difference between desktop and web applications**.

**Desktop applications**

* They must be developed for and installed on a particular operating system.
* Have strict hardware requirements that must be met to ensure that they function correctly.
* Updates to the applications must be applied by the user directly to their installation and may require hardware upgrades or other changes in order to work.

**Web applications**

* A web application is any computer program that performs a specific function by using a web browser.
* The user accesses the application using the web browser and works with resources available over the internet, including storage and CPU processing power.
* This approach allows for “thin clients” (machines with limited hardware capabilities) to provide access to complex applications delivered from a centralized infrastructure.

**c)Event oriented programming.**

In computer programming, event-driven programming is a programming paradigm in which the flow of the program is determined by events such as user actions (mouse clicks, key presses,

SetFocus, KeyDown,etc…),

**Event oriented programming using Visual Basic**

Visual Basic is Event oriented programming because of the following

* The programmer needs to write code that performs some tasks in response to certain events.
* Has events that occur by mouse clicking and moving or keyboard stokes (Some of the events are load, click, double click, drag and drop, pressing the keys and more.) Focus on the use of Graphical User Interface.
* The events usually comply but not limited to the user’s inputs.

**8.1.2 Standard EXE Visual Basic Application**

Using Visual Basic one can develop one of these main projects as indicated in the screen below:

**Standard Exe project** is a typical application in which can use the database manipulation.

A standard exe application is one that is created using Standard EXE project. It is the most widely used Project type using VB6. Standard EXE application is normally the most widely used among the available Project types in Visual Basic. Stand-alone programs have an .EXE file extension. A standard EXE application is normally used when you want to develop a stand-alone application.

Examples include calculators, text editors, and other similar applications.

**8.2Visual Basic Integrated Development Environment (VB-IDE)**

One of the most significant changes in Visual Basic 6.0 is the Integrated Development Environment (IDE). IDE is a term commonly used in the programming world to describe the interface and environment that we use to create our applications. It is called *integrated* because we can access virtually all of the development tools that we need from one screen called an *interface*.

The IDE is also commonly referred to as the ***design environment*.**

**The visual basic IDE is made up of the following components**

|  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
| |  |  | | --- | --- | | **a. Menu Bar** | **e. Tool Bar** | |  |  | | **b. Project Explorer** | **f. Properties Window** | |  |  | | **c. Form Layout Window** | **g. Tools Box** | |  |  | | **d. Form Designer** | **h. Object Browser** | |

**8.2.1Tool Box**

The Toolbox contains a set of controls that are used to place on a Form at design time thereby creating the user interface area. A Toolbox is represented in figure shown below

**8.2.2. Form window**

The form designer is the main window in the middle of the screen, in which we can design and edit user interface. The same window displays a text editor in which we can enter and edit applications code.

**8.2.3Code window or VB Editor**

Each standard form has a code window in which the user can write to direct the behavior of a control. You open the code window by double clicking on a form or a control. If you double click a form, you will be taken to a procedure for the form, but once the code window is open, you can go to any procedure for any object on the selected form.

The codes are of two Categories:

**Declaration** is written before any procedure in the code window.

**Statements**. The user selects the required event then code statements are written inside these event procedures.

**8.2.4Project explorer**

On the right side of the form design, just under the tool bar, is the Project Explorer window. The Project Explorer as shown in below figure serves as a quick reference to the various elements of a project namely *form*, *classes* and *modules*.

**8.2.5Properties Window**

Properties are the attribute of controls. Every object has properties, for example a Pen, has its Color, Metal Type, Ink Color, Type etc. In the same manner every control in VB has many properties which are applied by using the property fields in *property box*. Some properties are read only, which means the values of such properties can’t change using code, while others are Read and Write. You can move Property window any side of VB IDE Window, can appeal using tool bar or by Pressing F4.

The **Properties window** displays the properties of various controls and objects that are created in your applications. Each and every form in an application is considered an object.

**8.2.6Form Layout Window**

The Form Layout window is a visual design tool which is used to control the placement of the forms in the windows environment when they are executed especially when you have more than one form in your program,

To position a form in the Form Layout window, simply drag the miniature form to the desired location in the window.

**8.2.7. Menu Bar**

This is where you can select actions to perform on all your project files and to access help. When a project is open extra menus of project, build and data, are shown in addition to the default menu selection of File, edit, View, Debug, tools, window and Help.

**8.2.8. Toolbar**

It gives easy access to the menu-bar you use frequently.

**8.2.9 Object Browser**

Object browser allows us to browse through the Various Properties events and methods that are made available or exposed.

**8.3. Visual basic controls**

* + - * **A control:** is a tool you use to create objects on a Visual Basic form. You select controls from the toolbox and use the mouse to draw objects on a form. You use most controls to create user interface elements, such as command buttons, image boxes, and list boxes.
      * **Properties of a control.**

Before writing an event procedure for the control to respond to an event, you have to set certain properties for the control to determine its appearance and how it will work with the event procedure. You can set the properties of the controls in the properties window or at design time.

Other things you can do are to change its foreground and background color, change the font type and font size, enable or disable minimize and maximize buttons etc.

Here are the important points about setting up control properties:

* + - * Caption Property of a control should be clearly defined so that a user knows what to do with control
      * Use a meaningful name for property as it is easier to write and read the event procedure and easier to debug or modify the program later.
      * One more important property is whether to make the control enabled or not.
      * You should also consider making the control visible or invisible at design time, or when should it become visible or invisible.

The following are common controls used in Visual Basic:

**8.3.1 Form**

**Form** is used when you start Visual Basic, a default form (Form1) with a standard grid (a window consisting of regularly spaced dots) appears in a pane called the Form window. You can use the Form window grid to **create the user interface** and to line up interface elements.

**Steps to create Form**

**Step 1:** Click Project on menu and choose **Add form** from the list of options

**Step 2:** Click Form and click Open

**Step 3: Set the Properties for a form**

* + - * **Name:** Enter the name of the form. It returns the name used to identify form.
      * **Enabled:** form controls can be enabled or disabled on initial execution with the enabled property.
      * **Appearance:** It sets whether or not object is painted at run time with 3D effects.
      * **Caption:** Enter the caption associated with the form. It sets the text displayed as the title of the form.
      * **Backcolor:** Choose background color of textbox. It sets the background color used to display textbox.
      * **Borderstyle:** Choose border style of the form
      * **Fillcolor:** It sets the colour used to fill in shapes,circles and boxes.
      * **Fillstyle:** It returns the fill style of a shape
      * **Backstyle:** Choose background style; Opaque or transparent
      * **Font:** It returns the font of a form.
      * **Scaleheight:** It sets the number of units for the vertical measurement of form.
      * **Scaleleft:** It returns the horizontal coordinates for the left edge of form.
      * **Visible** shows or hide a control on a form. It sets value that determines whether an object is visible or hidden

**8.3.2. Label**

**Label** displays a text on the form. Labels are commonly used to identify the controls. The text in the label control can be changed by changing the caption property.

The following are the steps to add a label on the form.

**Step 1:** Click on the label icon on the toolbox

**Step 2:** Drag and drop a label control in the form

**Step 3:** Set Properties for label

* + - * **Name:** Enter the name of the label control. It returns the name used to identify object.
      * **Alignment**: It returns/sets the alignment of a label
      * **Appearance:** It sets whether or not object is painted at run time with 3D effects.
      * **Caption:** Enter the caption associated with the label. It sets the text displayed on the form.
      * **Font:** It returns the font of object (font face, font style, size and effects)
      * **Forecolor:** It returns foreground color of a label
      * **Height**: Enter the height of the label.
      * **Backcolor:** Choose background color. It sets the background color used to display label.
      * **Borderstyle:** Choose border style of the label(None-no border or Fixed Single-With border)
      * **Backstyle:** Choose background style; Opaque or transparent

**8.3.3. Text Box**

**Text box** is a box for entering and displaying text (characters or values) in user project. This tool is used frequently in most of the application.

**Steps to add Textbox**

**Step 1:** Click on the Textbox icon on the toolbox

**Step 2:** Drag and drop a Textbox control in the form

**Step 3:** Set Properties for Textbox such as Name, alignment, appearance, backcolor etc.

**8.3.4 Frame**

If you want to create a group of controls that work together, you must first create a frame for the controls. (To do this, use Frame, a Visual Basic toolbox control.) Next, place your controls inside the frame so that they can be processed as a group in your program code and moved as a group along with the frame

**Steps to add Frame**

Step 1: Click on the Frame icon on the toolbox

Step 2: Drag and drop a Frame control in the form

Step 3: Set Properties for Frame: on this step, you set preferred properties

* + 1. **Radio Buttons/ Option Buttons**

**Option Button control** is a part of an option group allows the user to select only one option even it displays multiple choices.

Unlike check boxes, selecting one option button immediately clears all the other buttons in the group.

**Steps to add Option Button**

**Step 1:** Click on the OptionButtonicon on the toolbox

**Step 2:** Drag and drop aOptionButton control in the form

**Step 3:** Set Properties for Option Button

* + 1. **Check box control**

Is similar to the **OptionButton**control, except that **Checkbox** displays a list of choices and gives the user the **option to pick multiple items (or none at all) from a listof choices**.

**Steps to add CheckBox**

**Step 1:** Click on the CheckBoxicon on the toolbox

**Step 2:** Drag and drop a CheckBox control in the form

**Step 3:** Set Properties for CheckBox

**8.3.6. Command button**

**Command button** Is a very important control as it is used to execute commands. It displays an illusion that the button is pressed when the user clicks on it. The most common event associated with the command button is the Click event.

**8.3.7 List Box**

**List Box** contains a list of options from which user can choose. In windows the **Font List box** is an example of the use of a list box. The Selected item in a ListBox is given by the *Text****property.*** The ***sorted*** property determines whether the items in the list box will be sorted or not.

**Adding items to list**

a) Change property list from properties window: Open the list properties window and choose

list property. Write a list of items to appear in the list box

**8.3.8 Combo Box**

The function of a Combo Box is to present a list of items where the user can click and select the items from the list. However, the user needs to click on the small arrowhead on the right of the combo box to see the items which are presented in a dropdown list.

**8.3.9. Picture Box**

**Picture Box** is a control used to display images on a VB page. The Picture Box control also supports few functionalities of generating advanced drawing.

**8.3.10. Imagebox**

**The Image Box**  is another control that handles images and pictures. It functions almost identically to the picture box. However, **there is one major difference, the image in an Image Box is stretchable, which means it can be resized**. This feature is not available in the Picture Box.

**8.3.11. Timer**

When we need to perform tasks at regular interval, we can use Timer **.** Open a new project and place a timer object on your form. Then locate a label at the center of the form and adjust the size as shown in the figure below. For such a program a better look will be established by sizing your form as a pop-up window.

**8.3.12. File System Controls**

**File System Controls** are set of controls which help us to add file handling capabilities to our program. They are used together to provide an interface for accessing and exploring drives, folders and files. File system controls can be drive list box, directory list box and file list box.

1. **Drive List Box:** The Drive ListBox is for displaying a list of drives available in your computer. When you place this control on the form and run the program, you will be able to select different drives from your computer.
2. **Directory List Box:**  is for displaying the list of directories or folders in a selected drive. When you place this control into the form and run the program, you will be able to select different directories from a selected drive in your computer.
3. **File List Box:**  This control displays a list of files in the current folder.

These are important controls even though there are plenty of controls used in VB.

**8.3.13. ADO**

**ADO** (ActiveX Data Objects) is the preferred method for accessing non-Jet tables. ADO and DAO are interchangeable for most things and you are unlikely to experience real timing differences with the two methods. There are some things that ADO cannot do with an Access database that DAO can do.DAO is native to Jet, and on several operations, it might perform a bit faster on Jet than ADO. DAO will contain some methods and properties that are only relevant to Jet, not other databases.

**8.3.14. DAO**

**DAO** (Data Access Objects) was the first object-oriented interface that exposed the Microsoft Jet database engine (used by Microsoft Access) and allowed Visual Basic developers to directly connect to Access tables as well as other databases - through Open Database Connectivity **(**ODBC).

**8.4.1 The process of planning and developing a visual basic program**

There are three steps involved in building a visual basic application:

* Draw the user interface
* Assign properties to controls
* Attach codes to control

1. **Draw the interface**

This step consists of designing the application’s user interface using controls. The controls are taken from the toolbox by dragging them from there to the form designer.

1. **Assign properties to control**

At this stage, user need to setup properties for the created form and controls. Those properties are set from the properties window.

1. **Write the event code.**

Coding is to be done in the code window and you get there by double clicking a form or control in the form.

Now we double click on the form1, the source code Window for the form1 appears as shown below:

Private Sub Form\_Load()

VB statement(s)

End Sub

You just have to write your code between the two statements. To display the output, you have to add Form\_name.show in order to display the statements or message written on the form.

**Example 1:** VB application to display the message: “**welcome to the world of programming:”**

Private sub Form\_load()

Form1.show

Print “Welcome to the world of programming”

End sub

**Example 2:** application to display date, time and message

Private Sub Form\_Load()

Form1.Show

Print " my message"

Print " today’s date is:" & Date

Print " and the time is: " & Time

Print " thanks and bye."

End Sub

**8.4.2. Debugging Your Code and Handling Errors**

**A. Debugging**

**Debugging** is a process by which you find and resolve errors in your code. To debug code in Visual Basic, consider the ideas suggested below. These techniques can also be applied in different sequences.

-Print the code, if you find it easier to read code on paper instead of softcopy.

-Run the application to find trouble spots:

From the Run **menu**, choose ‘Start’ to begin running the application. Run until an error stops execution, or halt execution manually when you suspect an error by choosing ‘Break’ from the Run menu. Resolve all compile errors and run-time errors. From the Run **menu**, choose ‘Continue’ to continue running the application.

**B. Handling errors**

Error handling should be used to process only exceptional situations, despite the fact that there is nothing to prevent that programmer from using errors as an alternate form of program control.

These errors can be grouped into three categories:

* Syntax errors
* Run-time errors
* Logic errors
* **Syntax errors** are grammatical errors in the formulation of statements and are picked up by the interpreter while you are typing in the code
* **Run-time errors** these are errors that cannot be detected until the program is running. The syntax of the statements is correct, but once executed they cause an error situation to arise. Examples of run-time errors are attempted division by zero or trying to access a non-existent object.
* **Logic errors** these are errors or bugs that cause the program to behave incorrectly. A logic error produces unintended or undesired output or other behavior, although it may not immediately be recognized. This is why careful testing is so important.

**8.4.3. Building an executable file**

You can make an executable file (.exe) from Visual Basic using the following procedure.

**To make an executable file in Visual Basic**

* 1. From the **File** menu, choose **Make** *projectname***.exe** where *projectname*is the application name for the project.
  2. Type a file name, or browse through the directories and select an existing file name to overwrite an existing executable with a newer version.
  3. By clicking the **Options** button, you can also specify a number of version-specific details about the executable file in the **Project Properties** dialog box.
  4. If you want to modify the version number of the project, set the appropriate **Major**, **Minor**, and **Revision** numbers. Selecting **Auto Increment** will automatically step

the **Revision** number each time you run the **Make** *projectname* **.exe** command for this project.

* 1. To specify a new name for the application, under **Application**, type a new name in the **Title** box. If you want to specify a new icon, choose one from the list.
  2. You can also enter version-specific commentary on a variety of issues under the **Version Information** box (comments, company name, trademark and copyright information, and so on) by selecting a topic from the list box and entering information in the text box.
  3. Choose **OK** to close the **Project Properties** dialog box, and then choose **OK** in the **Make** *appname* **.exe** dialog box to compile and link the executable file.

You can run the executable file like any other Windows-based application: double-click the icon for the executable file.

**8.5 Working with menus and dialog boxes**

## **8.5.1 Multiple Document Interface (MDI)**

The Multiple Document Interface (MDI) is one that allows you to view multiple windows within a larger window.

MDI differs from SDI (Single Document Interface) A single document interface is one where all windows appear independently of one another without the unification of a single parent window.

SDI applications allow only one open document frame window at a time.

Applications such as Microsoft office Word and Microsoft office excel are the best examples of MDI, where many documents can be opened simultaneously within the main document.

Each document is displayed in its own window and all document windows have the same behavior.

The main Form, or MDI form, isn’t duplicated, but it acts as a container for all the windows, and it is called the **parent Window**.

The windows in which the individual documents are displayed are called **Child windows.**

An MDI application must have at least two Form, the parent form and one or more child forms. Each of those forms has certain properties. These can be many child forms contained within the parent form, but there can be only one parent form.

To create an MDI application, at least two forms are needed in the application

One is the parent or container form and the second is the child form or the form contained within the parent.

A single child is required for the simple MDI Projects.

**Procedures**

* Select New project from the file menu
* In the case you already have a form, then in the properties window set the name property to frmchild and its caption to MDI\_Child
* Right-click on the forms folder in the project window
* Click on add from the MDI form to create the MDI Parent form.
* Select MDI form from the Add MDI form dialog
* In the Properties window set the name property to frmMDI and the caption property to MDI\_Parent.
* On the project menu, select project1.properties, set the startup object list to frmMDI. In the Case you omit this; the application will start with the child form.

Select the frmchild from the project Explorer

* Set the form’s MDI child property to true in the properties window. This will cause this form, which is the child, to rest inside the MDI Parent container.
* Save project as from the file menu and save the form as **MDI.frm** and project as **MDI.vbp**

**8.5.2 Creating Menus**

Menus, which are located on the **menu bar** of a form, contain a list of related commands. When you click a menu title in a Windows-based program, a list of menu commands should always appear in a well-organized list.

Most menu commands run immediately after they are clicked. For example, when the user clicks the **Edit** menu **Copy** command, Windows immediately copies information to the Clipboard.

However, if ellipsis points (…) follow the menu command, Visual Basic displays a dialog box that requests more information before the command is carried out.

This section includes the following topics:

* Using the Menu Editor
* Adding Access and Shortcut Keys Processing Menu Choices

**8.5.3 Using the Menu Editor**

The Menu Editor is a Visual Basic dialog box that manages menus in your programs. With the Menu Editor, you can:

* Add new menus
* Modify and reorder existing menus
* Delete old menus
* Add special effects to your menus, such as access keys, check marks, and keyboard shortcuts.

**Creating Menu Command Lists**

To build lists of menu commands, you first need to create the menus and then add them to the program menu bar.

**To create a list of menu commands on a form**

1. Click the form itself (not an object on the form).
2. On the Visual Basic toolbar, click the Menu Editor icon, or select **Menu Editor** from the **Tools** menu.
3. In the **Caption** text box, type the menu caption (the name that will appear on the menu bar), and then press TAB.
4. In the **Name** text box, type the menu name (the name the menu has in the program code). By convention, programmers use the *mnu*object name prefix to identify both menus and menu commands.
5. To add the menu to your program menu bar, click **next**. The Menu Editor clears the dialog box for the next menu item. As you build your menus, the structure of the menus and commands appear at the bottom of the dialog box.
6. In the **Caption** text box, type the caption of your first menu command.
7. Press tab, and then type the object name of the command in the **Name** text box.
8. With this first command highlighted in the menu list box, click the right arrow button in the Menu Editor. In the **Menu** list box, the command moves one indent (four spaces) to the right. Click the right arrow button in the Menu Editor Dialog box to move items to the right, and click the left arrow button to move items to the left.
9. Click **Next**, and then continue to add commands to your menu.

The position of list box items determines what they are:

|  |  |
| --- | --- |
| List box item | Position |
| Menu title | Flush left |
| Menu command | One indent |
| Submenu title | Two indents |
| Submenu command | Three indents |

**To add more menus**

1. When you are ready to add another menu, click the left arrow button to make the menu flush left in the **Menu** list box.
2. To add another menu and menu commands, repeat Steps 3 through 9 in the preceding procedure. 3. When you are finished entering menus and commands, click **OK** to close the Menu Editor.

(Don’t accidentally click **Cancel** or all your menu work will be lost.) The Menu Editor closes, and your form appears in the programming environment with the menus you created.

Adding Event Procedures After you add menus to your form, you can use event procedures to process the menu commands. Clicking a menu command on the form in the programming environment displays the event procedure that runs when the menu command is chosen.

**8.5.4 Adding Access and Shortcut Keys.**

Visual Basic makes it easy to provide access key and shortcut key support for menus and menu commands.

**Access and Shortcut Keys**

The access key for a command is the letter the user can press to execute the command when the menu is open. The shortcut key is the key combination the user can press to run the command without opening the menu. Here's a quick look at how to add access and shortcut keys to existing menu items:

Add an access key to a menu item Start the Menu Editor. Prefix the access key letter in the menu item caption with an **ampersand (&).**

Add a shortcut key to a menu command Start the Menu Editor. Highlight the command in the menu list box. Pick a key combination from the Shortcut drop-down list box.

**Creating Access and Shortcut Keys**

You can create access keys and shortcut keys either when you first create your menu commands or at a later time.

The following illustration shows the menu commands associated with two menus, **File** and **Clock**. Each menu item has an access key ampersand character, and the **Time** and **Date** commands are assigned shortcut keys. See figure below.

**8.5.4 Processing Menu Choices**

After you place menu items on the menu bar, they become objects in the program. To make the menu objects do meaningful work, you need to write event procedures for them. Typically, menu event procedures:

* Contain program statements that display or process information on a form.
* Modify one or more object properties.

For example, the event procedure for a command named **Time** might use the **Time** keyword to display the current system time in a text box.

Processing the selected command might require additional information (you might need to open a file on disk, for example). If so, you can display a dialog box to receive user input by using a common dialog box. You’ll learn this technique in the next section.

**Disabling a Menu Command**

In a typical Windows application, not all menu commands are available at the same time. In a typical **Edit** menu, for example, the **Paste** command is available only when there is data on the Clipboard. When a command is disabled, it appears in dimmed (gray) type on the menu bar. You can disable a menu item by:

* Clearing the **Enabled** check box for that menu item in the Menu Editor.
* Using program code to set the item's Enable property to False. (When you’re ready to use the menu command again, set its Enable property to True.).