VIDYAPITH ACADEMY

A unit of AITDC (OPC) PVT. LTD.

IAF Accredited An ISO 9001:2015 Certified Institute.

Registered Under Ministry of Corporate Affairs

(CIN U80904AS2020OPC020468)

Registered Under MSME, Govt. of India. (UAN- AS04D0000207).

Registered Under MHRD (CR act) Govt. of India.

DIPLOMA IN COMPUTER SOFTWARE

TOPIC 1: OBJECT ORIENTED PROGRAMMING USING C & C++
TOPIC 2: OPERATING SYSTEM
TOPIC 3: COMPUTER ARCHITECTURE
TOPIC 4: PROGRAMMING USING VISUAL BASIC
TOPIC 5: DATA STRUCTURE
PRACTICAL LAB ASSIGNMENT & VIVA VOICE

OBJECT ORIENTED PROGRAMMING USING C & C++

Why we use Language ?

We use language due to following reasons : ➢
It stores data
- ➢ How to interact between input and output
- ➢ The operator uses to transform and combine data.

What is C ?

By 1960 languages come into existence. COBOL was being used for commercial purposes, FORTRAN for Engineering and Scientific applications. An International committee was setup to develop languages. This committee came out with a language called ALGOL60. ALGOL-60 never really became popular because it seemed too abstract, too general. To reduce the abstractness and generality a new language called Combined Programming Language (CPL) was developed at Cambridge University.

CPL was an attempt to bring ALGOL 60 new version, but CPL turned out to be so big, having so many features, that it was hard to learn and difficult to implement. BCPL (Basic Combined Programming Language) developed by Martin Richards at Cambridge University aimed to solve this problem by bringing CPL down to its basic good features. But unfortunately it turned to be less powerful and too specific. Dennis Ritchie inherit the features of CPL and BCPL and made a language named C.

C forms the basis for many advanced, highly powerful and effective programming languages. C is a programming language developed at AT&T‘s Bell Laboratories of USA in 1972. It was designed and written by a man named Dennis Ritchie. In the late seventies C began to replace the more familiar languages of that time like PL/I, ALGOL etc. No one pushes C. C seems so popular, because it is reliable, simple and easy to use. The concept of C derives its origin from a primitive form of C which was called Basic Combined Programming Language (BCPL) developed by Ken Thompson of Bell Laboratories which he referred to as ‗B‘ . By now it might not be difficult to guess that a name as cryptic as C was conferred to it because it was considered to be a modified more adaptive successor of ―B‖. C‘s compactness and coherence is mainly due to the fact that it‘s a one man language. **C Basics**

Before we embark on a brief tour of C's basic syntax and structure we offer a brief history of C and consider the characteristics of the C language.
In the remainder of the Chapter we will look at the basic aspects of C programs such as C program structure, the declaration of variables, data types and operators. We will assume knowledge of a high level language, such as PASCAL.

It is our intention to provide a quick guide through similar C principles to most high level languages. Here the syntax may be slightly different but the concepts exactly the same.

C does have a few surprises:

- Many High level languages, like PASCAL, are highly disciplined and structured.
- However beware -- C is much more flexible and free-wheeling. This freedom gives C much more power that experienced users can employ. The above example below (mystery.c) illustrates how bad things could really get.

Characteristics of C

We briefly list some of C's characteristics that define the language and also have lead to its popularity as a programming language. Naturally we will be studying many of these aspects throughout the course.

- Small size
- Extensive use of function calls
- Loose typing -- unlike PASCAL
- Structured language
- Low level (BitWise) programming readily available

- Pointer implementation - extensive use of pointers for memory, array, structures and functions.

C has now become a widely used professional language for various reasons.

- It has high-level constructs.
- It can handle low-level activities.
- It produces efficient programs.
- It can be compiled on a variety of computers.

Features of C	
Low Level Language Support	Program Portability
Powerful and Feature Rich	Bit Manipulation
High Level Features	Modular Programming
Efficient Use of Pointers	

Features of C Programming Language :

C Programming is widely used in Computer Technology, We can say that C Programming is inspiration for development of other languages. We can use C Programming for different purposes. Below are some of the Features of C Programming language -

1 . Low Level Features :

1. C Programming provides low level features that are generally provided by the Lower level languages. C is Closely Related to Lower level Language such as ¯**Assembly Language**¯.
2. It is easier to write assembly language codes in C programming.

2 . Portability :

1. C Programs are portable i.e they can be run on any Compiler with Little or no Modification
2. Compiler and Preprocessor make it Possible for C Program to run it on Different PC

3 . Powerful

1. Provides Wide verity of _**Data Types**_
2. Provides Wide verity of _Functions'
3. Provides useful Control & Loop Control Statements

4 . Bit Manipulation

1. C Programs can be manipulated using bits. We can perform different operations at bit level. We can manage memry representation at bit level.
2. It provides wide verity of bit manipulation Operators. We have bitwise operators to manage Data at bit level.

5 . High Level Features :

1. It is more User friendly as compare to Previous languages. Previous languages such as BCPL,Pascal and other programming languages never provide such great features to manage data.
2. Previous languages have there **pros and cons** but C Programming collected all useful features of previous languages thus C become **more effective language**.

6 . Modular Programming

1. **Modular programming** is a software design technique that increases the extent to which software is composed of separate parts, called **modules**
2. C Program Consist of Different Modules that are integrated together to form complete program

7 . Efficient Use of Pointers

1. Pointers has direct access to memory.
2. C Supports efficient use of pointer .

C Program Structure

A C program basically has the following form:

- Preprocessor Commands
- Type definitions
- Function prototypes -- declare function types and variables passed to function.
- Variables
- Functions

We must have a main() function.

C Keywords

Keywords are the set of predefined words whose functionality has been expressed to the compiler and whenever called upon they furnish their task with utmost comfort. The keywords cannot be used for any function other than what it is defined for, not even as variable names.

Variables

A program is made of data and instructions to manipulate those data. Note that data have to be stored somewhere, and thus will need some memory space in the RAM.
A variable is an entity that is used to store data. Without variables, there is no way (or actually NO PLACE) to store data. A variable has

- a name (more specifically a symbolic name)
- an associated physical memory space (portion in a RAM)
- a data type
- a value (depends on data type)
- a scope
- a lifetime

Basic Data Types

The language of C support several types of Data, each of which is represented in a varied manner with in the memory. The data in the memory can be of integer type or of character type or of integer with decimal points. Whenever an integer or a character is used in a program the computer should be able to identify where to store it in the memory.
The basic data types are depicted in a tabular format to get a better understanding of the basics of C programming.

Data Type	Description	Memory Requirement	Range	Format Specifier

Int	whole numbers	2 bytes	-32768 to	%d, %i
long	-	4 bytes	+32767	%ld
Char	Characters	1 Byte	0 to 255	%c, %s
Float	Numbers with Decimals	4 Bytes	1.0E-37 to 1.0E+37	%f
Double	Numbers with	8 Bytes	1.7E-308 to	%lf

Constants

ANSI C allows you to declare **constants**. When you declare a constant it is a bit like a variable declaration except the value cannot be changed. The const keyword is to declare a constant, as shown below:

int const a = 1; const
int a =2;

Note:

- You can declare the const before or after the type. Choose one an stick to it.
- It is usual to initialise a const with a value as it cannot get a value **any other way**.

The preprocessor #define is another more flexible (see Preprocessor Chapters) method to define **constants** in a program.You frequently see const declaration in function parameters. This says simply that the function is **not** going to change the value of the parameter.The following function definition used concepts we have not met (see chapters on functions, strings, pointers, and standard libraries) but for completenes of this section it is is included here:

void strcpy(char *buffer, char const *string)

The second argiment string is a C string that will not be altered by the string copying standard library function.

Operators in C :

Operators can briefly be defined as the tools used for solving various mathematical, conditional, relational and logical problems. The operators that you will be dealing with are arithmetic operators , unary operators, relational and logical operators . The item in between which the operators are placed are called operands.

Arithmetic operators :

Arithmetic Operators can be considered the main point of all the operators. Arithmetic operators are tools that help us in computing various mathematical operations.

Operators	**Function Performed**
+	Addition
	Subtraction
*	Multiplication
/	Division
%	(Modulus) Finds the remainder

Unary Operators :

There are two types of unary operators :-

- **The increment operator (denotes as ‘ ++ ’)**
- **The decrement operator (denotes as ‘ -- ’)**

When the increment operator is prefixed to a variable which holds an integer it increases the value of the number by one. Similarly the decrement operator decreases the value by one.

Relational and Logical Operators :

Operators	**Significance**
<	Less than
<=	Less than or equal to
>	Greater than
>=	Greater than or equal to
Equality Operators	Significance
==	Equal to
! =	Not equal to
Logical Operators	Significance
&&	And
| |	Or
!	Not

Assignment is = ***i.e.*** i = 4; ch = `y';

Increment ++, **Decrement** -- which are more efficient than their long hand equivalents, for example:-- x++ is faster than x=x+1.

The ++ and -- operators can be either in post-fixed or pre-fixed. With pre-fixed the value is computed before the expression is evaluated whereas with post-fixed the value is computed after the expression is evaluated.

Comparison Operators

To test for equality is ==

A warning: Beware of using ``='' instead of ``=='', such as writing accidentally

```
if ( i = j ) .....
```

This is a perfectly **LEGAL** C statement (syntactically speaking) which copies the value in "j" into "i", and delivers this value, which will then be interpreted as TRUE if j is nonzero. This is called **assignment by value** -- a key feature of C.

Not equals is: !=

Other operators < (less than) , > (grater than), <= (less than or equals), >= (greater than or equals) are as usual.

Statements in C Language

C needs some kind of managerial system which instructs the computer what function has to be performed at which time. The control statement act as a manager. You will amazed at the functions the different control statements handle. For example in a program you may have a statement in which a multiplication operation is to be performed and in the next an addition operation and after the next line there may be a statement which may want itself to be repeated 8 times and another statement may like itself to be skipped and attended to at a later stage. All these conditions are managed by the control statements.

The sequence control structure is one of the basic uncomplicated control statements. The idea is very simple. In the sequence control instructions the statements to be executed are done so in a serial manner one after the other. The C compiler first executes the opening line and then moves on to the next and so on. In a serial fashion and the entire program is executed in this manner.

Conditional Statement

C program executes program sequentially. Sometimes, a program requires checking of certain conditions in program execution. C provides various key condition statements to check condition and execute statements according conditional criteria.These statements are called as '**Decision Making Statements**' or '**Conditional Statements**'.Followings are the different conditional statements used in C :

1. **If Statement**
2. **If-Else Statement**
3. **Nested If-Else Statement**
4. **Switch Case**

If Statement: The if statement can be used to test conditions so that we can alter the flow of a program. Code:

```
#include<stdio.h>
int main()
{
        int mark; char pass;
        scanf("%d",&mark);
        if (mark > 40) pass
        = "y";
                return 0;
}
```

If-Else Statement: The if statement first tests if a condition is true and then executes an instruction and the else is for when the result of the condition is false. Code:

```
#include<stdio.h> int
main()
{
        int mark; char pass;
        scanf("%d",&mark);
        if (mark > 40)
        {
                pass = "y";
                printf("You passed");
        }
        else
        {
                pass = "n"; printf("You
                failed");
        }
        return 0;
}
```

The Switch Statement: The switch statement is just like an if statement but it has many conditions and the commands for those conditions in only 1 statement. It is runs faster than an if statement. In a switch statement you first choose the variable to be tested and then you give each of the conditions and the commands for the conditions. You can also put in a default if none of the conditions are equal to the value of the variable. **For Loop, While Loop, Break and Continue**

Control structures are basically of three types -

- **Sequence statements**
- **Iterative statements**
- **Selection statements**

Sequence Statements : All the State in a program except the iterative & statements. They are generally the individual statements which performs the task of input, output, assignment declaration etc.

Iterative Statement are those repeated execution of a particular set of instructions desired number of times. These statements are generally called loops for their execution nature.

Types of Looping Statements:

Basically, the types of looping statements depends on the condition checking mode. Condition checking can be made in two ways as : Before loop and after loop. So, there are 2(two) types of looping statements.

- **Entry controlled loop** 	**Exit controlled loop**

1. Entry controlled loop :

In such type of loop, the test condition is checked first before the loop is executed.

Some common examples of this looping statements are :

- **while loop**
- **for loop**

2. Exit controlled loop :

In such type of loop, the loop is executed first. Then condition is checked after block of statements are executed. The loop executed atleat one time compulsarily.

Some common example of this looping statement is :

- **do-while loop**

In C language the iterative statements (loops) can be implemented in the three loops and they are

The For Loop

Syntax-

```
for (initialization; condition; incrementation)
{
  ---------------- body of loop ------------
}
```

For Loop will perform its execution until the condition remains satisfied. If the body of the loop consists of more than one statement then these statements are made compound by placing the open and closed curly brackets around the body of the loop. For loop is a count loop. The initialization condition and increamentation may be done in the same statement. For loop will not execute at least once also if the condition is false at the first time itself.

The While Loop

Syntax –

```
        Initialization;
    While (condition)
```

```
{
 Body of loop;
         Incrementation;
}
```

In this loop, initialisation, condition and incrementation is done in the three different statements. This loops is count as well as event loop. In case of while loops the body of the loop will consist of more than one statements because each time one statement will be of incrementation. Hence the open and closed curly brackets are required.

Do-While Loop Statement:

The third loop statement available in C is do-while statement syntax :-

```
Initialization;
Do
{
Body of loop;
Increamentation;
}while(condition)
```

An introduction to Arrays

The concept of arrays depends on other data types, which was meant to facilitate the storage of abundant amount of number and wasting an entire delivery of space in the memory. It would be like buying a two dozen books and ultimately using one and dumping the rest in the garbage. Arrays can be defined as a collection of similar elements. An array of elements can be formed only if all the elements are of one particular data type i.e. all of them are either integer or character or a floating point number, but there cannot exist an array which hosts an combination of these data types.

Types of Arrays

The array are divided in to two parts :

- **Single Dimensional Arrays**
- **Multidimensional Arrays**

Single Dimensional Arrays :

The single dimensional arrays as the name suggested, handles only a single advance of similar elements. That is to say you can only have one row of elements and the size of a single dimensional array depends on the programmer. Basically the single dimensional arrays facilitate the accommodation of many similar elements in a single variable. The succeeding sections after which you will have a cloud less view of the single dimensional arrays. **Multi Dimensional Arrays :**

Multi dimensional array requires separate brackets for each subscript. One dimensional array has a pair of square brackets, a two dimensional array will have two pairs of square brackets, three dimensional array three pairs of square brackets and so on.

Two Dimensional Array :

A two dimensional array can be visualized as an array below an array. Two dimensional array will have two pairs of square brackets.
Syntax :- datatype arrayname [statement 1] [statement 2]
The two dimensional array is often referred to as a matrix. The statement one and two are the subscripts that the two dimensional array will hold. One will specify the row and expression, two will specify the column and they indicate the number of array elements associated with each subscript.
int A [4] [2] ;

	Column 0	Column 1
Row No 0	4200	10
Row No 1	6500	20
Row No 2	7850	30
Row No 3	3450	40

Thus the element in A [0][0] will be 1000 and the element A[3][3] will be 40. The rest of the numbers can be figured by having a look at the above table. In the memory the element are placed in a sequential pattern such that the element [0][0] will be first followed by the element in [0][1] followed by [1][0] and so on.

Strings

A group of integers can be stores in an integer array. Similarly a group of characters can be stored in a character array. Character array are many a time also called strings. Most languages internally treat strings as character arrays, but some how conceal this fact from the programmer. Character arrays or strings are used by programming languages to manipulate text such as words and sentences. A string constant is a one dimensional array of characters terminated by a null (‗ \ 0 ') for example ,
char name [ ] = { ‗ I ', ‗ T ' , ' T ' , ' C ' , ‗ O ' , ‗ M ' , ' P ' , ' U ' , ‗ T ' , ‗ E ' , ‗ R ', ‗ \0 '}
Each character in the array occupies one byte of memory and the last character is always ‗ \ 0 '. What character is this ? It looks like two characters, but it is actually only one character, either the **** indicating that what follows it is something special. ‗ \ 0 ‗ is called null character. Note that ‗\0' and ‗0' are not same. ASCII value of ‗ \0 ‗ is 0 whereas ASCII value of ‗ 0 ‗ is 48 It shows a way a character array is stored in memory. The elements of character array are stored in contiguous memory locations.
The terminating null (‗ \ 0 ‗) is important, because it is the only way the functions that work with a string can know where the string can know where the string ends. A string not terminated by a ‗ \0 ‗ is really a string, but merely a collection of characters.

With C compiler a large set of useful string handling library functions are provided.

Function	**Use**
strlen	Finds length of a string
strlwr	Converts a string to lowercase
strupr	Converts a String to Upper Case
strcat	Appends one string at the end of another

strcpy	Copies a String into Another
strcmp	Compares two strings
strdup	Duplicating a String
strrev	Reversing a String

Function

A function is a self contained block of a statements that perform a coherent task of some kind. Every C program can be thought of as a collection of these functions.

Sometimes the interaction with this person is very simple sometimes it‘s complex. You have a task which is always performed exactly in the same way... say a servicing of your motorbike. When you want it to be done, you go to the service station and say ¯ I‘s time, do it now You don‘t need to give instructions, because the mechanic knows his job. You don‘t need to be told when the job is done. U assume the bike would be serviced in the usual way, the mechanic does.
A simple C function which operated in much the same way as the ,mechanics. We will be looking at two things a program that calls or activates the function and the function itself.

```
void main ( )
    {       message ( ) ;
             printf  (¯  \n Thanks after return of first function ¯ ) ;
             getch ( ) ;
    }
    message ( )
    {
     printf  ( ¯ \n  Welcome to the first function program  ¯ ) ;
    }
```

Note :
Any C program contains at least one functions.
If a program contains only one function, it must be **main ( )** .
In a C program if there are more than one functions present, then one (and only one) of these functions must be main( ), because program execution always begins with **main ( )** .
There is no limit on the number of functions that might be presented in a C program. Each function in a program is called in the sequence specified by the function calls in **main( )**. After each function has done its things, control returns to **main ( )**. When **main ( )** runs out of function calls, the program ends.

Language C++

Introduction

object-orientation is introduced as a new programming concept which should help you in developing high quality software. Some people will say that object-orientation is ``modern''. When reading announcements of new products everything seems to be ``object-oriented''. ``Objects'' are everywhere. In this section we will try to outline characteristics of object-orientation to allow you to judge those object-oriented products.

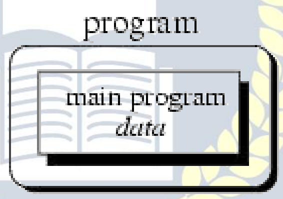
Roughly speaking, we can distinguish the learning curve of someone who learns to program:

- Unstructured programming,
- procedural programming,
- modular programming and object-oriented programming.

Unstructured Programming

Usually, people start learning programming by writing small and simple programs consisting only of one main program. Here ``main program'' stands for a sequence of commands or *statements* which modify data which is *global* throughout the whole program.

Figure: Unstructured programming. The main program directly operates on global data.



Procedural Programming

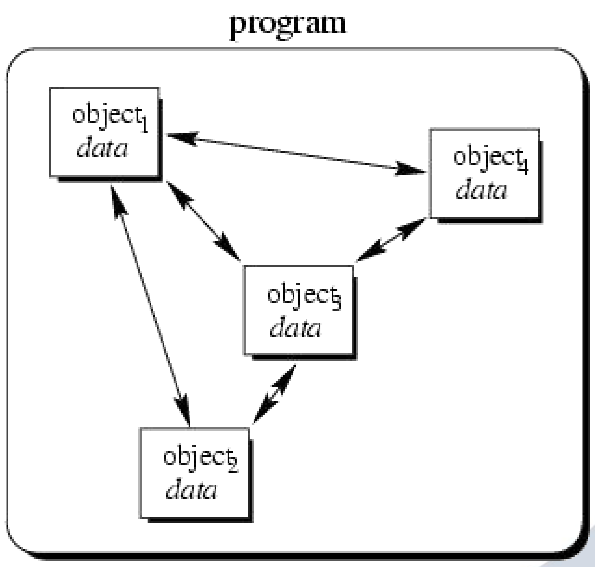
With procedural programming you are able to combine returning sequences of statements into one single place. A *procedure call* is used to invoke the procedure. After the sequence is processed, flow of control proceeds right after the position where the call was made

Modular Programming

With modular programming procedures of a common functionality are grouped together into separate *modules*. A program therefore no longer consists of only one single part. It is now devided into several smaller parts which interact through procedure calls and which form the whole program.

Object-oriented programming solves some of the problems just mentioned. In contrast to the other techniques, we now have a web of interacting objects, each house-keeping its own state.

Figure: Object-oriented programming. Objects of the program interact by sendin messages to each other.



Abstract Data Types

Some authors describe object-oriented programming as programming *abstract data types* and their relationships. ADTs are used to define a new type from which instances can be created. ADTs provide an abstract view to describe properties of sets of entities, their use is independent from a particular programming language.

Class

A class is an actual representation of an ADT. It therefore provides implementation details for the data structure used and operations. We play with the ADT Integer and design our own class for it: class Integer { attributes:

```
   int i

 methods:
   setValue(int n)
   Integer addValue(Integer j)
 }
```

A class is the implementation of an abstract data type (ADT). It defines attributes and methods which implement the data structure and operations of the ADT, respectively. Instances of classes are called *objects*. Consequently, classes define properties and behaviour of sets of objects.

Object

Objects are uniquely identifiable by a *name*. Therefore you could have two distinguishable objects with the same set of values. This is similar to ``traditional'' programming languages where you could have, say two integers *i* and *j* both of which equal to ``2''. Please notice the use of ``i'' and ``j'' in the last sentence to name the two integers. We refer to the set of values at a particular time as the *state* of the object.

Definition (Object) An object is an instance of a class. It can be uniquely identified by its name and it defines a state which is represented by the values of its attributes at a particular time. The ***behaviour*** *of an object is defined by the set of methods which can be applied on it.*

Message

A running program is a pool of objects where objects are created, destroyed and *interacting*. This interacting is based on *messages* which are sent from one object to another asking the recipient to apply a method on itself. A **message** is a request to an object to invoke one of its methods. A message therefore contains

- the **name** of the method and
- the **arguments** of the method.

Inheritance

With inheritance we are able to make use of the a-kind-of and is-a relationship. As described there, classes which are a-kind-of another class share properties of the latter. In our point and circle example, we can define a circle which *inherits from* point:

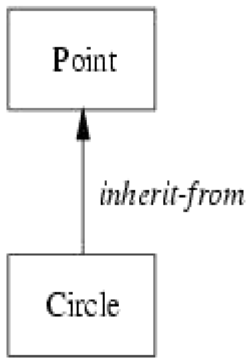
```
  class Circle inherits from Point
{attributes:    int radius

  methods:     setRadius(int
newRadius)     getRadius()}
```

Class *Circle* inherits all data elements and methods from point. There is no need to define them twice: We just use already existing and well-known data and method definitions.

Definition (Inheritance) Inheritance is the mechanism which allows a class A to inherit properties of a class B. We say ``A inherits from B''. Objects of class A thus have access to attributes and methods of class B without the need to redefine them. The following definition defines two terms with which we are able to refer to participating classes when they use inheritance.

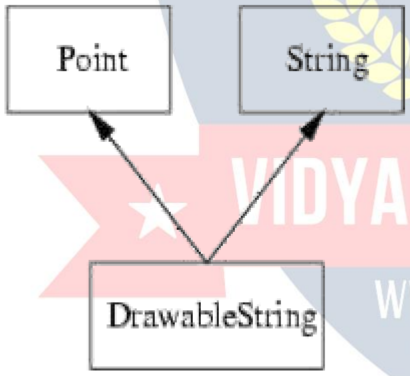
Definition (Superclass/Subclass) If class A inherits from class B, then B is called superclass of A. A is called subclass of B. Objects of a subclass can be used where objects of the corresponding superclass are expected. This is due to the fact that objects of the subclass share the same behaviour as objects of the superclass.

Figure: A simple inheritance graph.



Multiple Inheritance
One important object-oriented mechanism is multiple inheritance. Multiple inheritance does **not** mean that multiple subclasses share the same superclass. It also does **not** mean that a subclass can inherit from a class which itself is a subclass of another class.Multiple inheritance means that one subclass can have *more than one* superclass. This enables the subclass to inherit properties of more than one superclass and to ``merge'' their properties.

Figure: Derive a drawable string which inherits properties of Point and String.



What is inherited from the base class?

In principle, a derived class inherits every member of a base class except:

- its constructor and its destructor
- its operator=() members
- its friends

Although the constructors and destructors of the base class are not inherited themselves, its default constructor (i.e., its constructor with no parameters) and its destructor are always called when a new object of a derived class is created or destroyed.

Abstract Classes

With inheritance we are able to force a subclass to offer the same properties like their superclasses. Consequently, objects of a subclass behave like objects of their superclasses. Sometimes it make sense to only describe the properties of a set of objects without knowing the actual behaviour beforehand.

Static and Dynamic Binding

In strongly typed programming languages you typically have to *declare* variables prior to their use. This also implies the variable's *definition* where the compiler reserves space for the variable. *Definition (Static Binding) If the type T of a variable is explicitly associated with its name N by declaration, we say, that N is statically bound to T. The association process is called static binding.*

Polymorphism

Polymorphism allows an entity (for example, variable, function or object) to take a variety of representations. Therefore we have to distinguish different types of polymorphism which will be outlined here.

Data Types

Table: Built-intypes.

Type	Description	Size	Domain
char	Signed character/byte. Characters are enclosed in **single** quotes.	1	-128..127
double	Double precision number	8	ca. $10^{-308}..10^{308}$
int	Signed integer	4	$-2^{31}..2^{31}-1$
float	Floating point number	4	ca. $10^{-38}..10^{38}$
long (int)	Signed long integer	4	$-2^{31}..2^{31}-1$
long long (int)	Signed very long integer	8	$-2^{63}..2^{63}-1$
short (int)	Short integer	2	$-2^{15}..2^{15}-1$
unsigned char	Unsigned character/byte	1	0..255
unsigned (int)	Unsigned integer	4	$0..2^{32}-1$
unsigned long (int)	Unsigned long integer	4	$0..2^{32}-1$
unsigned long long (int)	Unsigned very long integer	8	$0..2^{64}-1$
unsigned short (int)	Unsigned short integer	2	$0..2^{16}-1$

Expressions and Operators
Expressions are combined of both *terms* and *operators*. The first could be constansts, variables or expressions. From the latter, C offers all operators known from other languages. However, it offers some operators which could be viewed as abbreviations to combinations of other operators. In C almost everything is an expression. For example, the assignment statement ``='' returns the value of its righthand operand. As a ``side effect'' it also sets the value of the lefthand operand.

Statements

C defines all usual flow control statements. Statements are terminated by a semicolon ``;''. We can group multiple statements into blocks by enclosing them in curly brackets. Within each block, we can define new variables

Statement	Description
`break;`	Leave current block. Also used to leave `case` statement in `switch`.
`continue;`	Only used in loops to continue with next loop immediately.

`do`
stmt
`while (`*expr*`);`
	Execute *stmt* as long as *expr* is TRUE.

`for ([`*expr*`]; [`*expr*`]; [`*expr*`])` *stmt*	This is an abbreviation for a `while` loop where the first *expr* is the initialization, the second *expr* is the condition and the third *expr* is the step.
`goto` *label*;	Jumps to position indicated by *label*. The destination is *label* followed by colon ":".
`if (`*expr*`)` *stmt* [`else` *stmt*]	IF-THEN-ELSE in C notation
`return` [*expr*];	Return from function. If function returns `void return` should be used without additional argument. Otherwise the value of *expr* is returned.

`switch (`*expr*`) {`
`case` *const-expr*: *stmts*
`case` *const-expr*: *stmts*
...
[`default:` *stmts*]
}
	After evaluation of *expr* its value is compared with the `case` clauses. Execution continues at the one that matches. BEWARE: You **must** use `break` to leave the `switch` if you don't want execution of following `case` clauses! If no `case` clause matches and a `default` clause exists, the statements of the default clause are executed.

`while (`*expr*`)` *stmt*	Repeat *stmt* as long as *expr* is TRUE.

Functions

As C is a procedural language it allows the definition of *functions*. Procedures are ``simulated'' by functions returning ``no value''. This value is a special type called void.

Functions are declared similar to variables, but they enclose their arguments in parenthesis (even if there are no arguments, the parenthesis must be specified):

```
  int sum(int to);  /* Declaration of sum with one argument */
int bar();        /* Declaration of bar with no arguments */
void foo(int ix, int jx);
                  /* Declaration of foo with two arguments */
```

Pointers and Arrays

One of the most common problems in programming in C++ is the understanding of pointers and arrays. In C (C++) both are highly related with some small but essential differences. You declare a pointer by putting an asterisk between the data type and the name of the variable or function:

```
  char *strp;     /* strp is `pointer to char' */
```

You access the content of a pointer by dereferencing it using again the asterisk:

```
  *strp = 'a';           /* A single character */
```

As in other languages, you must provide some space for the value to which the pointer points. A pointer to characters can be used to point to a sequence of characters: the *string*. Strings in C are terminated by a special character NUL (0 or as char ',\0).

Constructor

With constructors we are able to initialize our objects at definition time as we have requested it for our singly linked list. We are now able to define a class *List* where the constructors take care of correctly initializing its objects.If we want to create a point from another point, hence, copying the properties of one object to a newly created one, we sometimes have to take care of the copy process. For example, consider the class *List* which allocates dynamically memory for its elements. If we want to create a second list which is a copy of the first, we must allocate memory and copy the individual elements.

Destructors

Consider a class List. Elements of the list are dynamically appended and removed. The constructor helps us in creating an initial empty list. However, when we leave the scope of the definition of a list object, we must ensure that the allocated memory is released. We therefore define a special method called destructor which is called once for each object at its destruction time:

```
  void foo() {
        List alist;       // List::List() initializes to
// empty list.
    ...                   // add/remove elements   }                    //
Destructor call!
```

Destruction of objects take place when the object leaves its scope of definition or is explicitly destroyed. The latter happens, when we dynamically allocate an object and release it when it is no longer needed. Destructors are declared similar to constructors.

Thus, they also use the name prefixed by a tilde (~) of the defining class. **Operators in C++**

Operators can briefly be defined as the tools used for solving various mathematical, conditional, relational and logical problems. The operators that you will be dealing with are arithmetic operators , unary operators, relational and logical operators . The item in between which the operators are placed are called operands.

Arithmetic operators :

Arithmetic Operators can be considered the main point of all the operators. Arithmetic operators are tools that help us in computing various mathematical operations.

Operators	**Function Performed**
+	Addition
	Subtraction
*	Multiplication
/	Division
%	(Modulus) Finds the remainder

Unary Operators :

There are two types of unary operators :-

- **The increment operator (denotes as ‘ ++ ’)**
- **The decrement operator (denotes as ‘ -- ’)**

When the increment operator is prefixed to a variable which holds an integer it increases the value of the number by one. Similarly the decrement operator decreases the value by one.

Relational and Logical Operators :

Operators	**Significance**
<	Less than
<=	Less than or equal to
>	Greater than
>=	Greater than or equal to
Equality Operators	Significance
==	Equal to
! =	Not equal to
Logical Operators	Significance

&&	And
| |	Or
!	Not

Assignment is = ***i.e.*** i = 4; ch = `y';

Increment ++, **Decrement** -- which are more efficient than their long hand equivalents, for example:-- x++ is faster than x=x+1.

The ++ and -- operators can be either in post-fixed or pre-fixed. With pre-fixed the value is computed before the expression is evaluated whereas with post-fixed the value is computed after the expression is evaluated.

Comparison Operators

To test for equality is ==

A warning: Beware of using ``='' instead of ``=='', such as writing accidentally

```
  if ( i = j ) .....
```

This is a perfectly **LEGAL** C statement (syntactically speaking) which copies the value in "j" into "i", and delivers this value, which will then be interpreted as TRUE if j is nonzero. This is called **assignment by value** -- a key feature of C.

Not equals is: !=

Other operators < (less than) , > (grater than), <= (less than or equals), >= (greater than or equals) are as usual.

Statements in C Language

C needs some kind of managerial system which instructs the computer what function has to be performed at which time. The control statement act as a manager. You will amazed at the functions the different control statements handle. For example in a program you may have a statement in which a multiplication operation is to be performed and in the next an addition operation and after the next line there may be a statement which may want itself to be repeated 8 times and another statement may like itself to be skipped and attended to at a later stage. All these conditions are managed by the control statements.

. **Conditional Statement**

C program executes program sequentially. Sometimes, a program requires checking of certain conditions in program execution. C provides various key condition statements to check condition and execute statements according conditional criteria.These statements are called as '**Decision Making Statements**' or '**Conditional Statements**'.Followings are the different conditional statements used in C :

5. **If Statement**
6. **If-Else Statement**
7. **Nested If-Else Statement**
8. **Switch Case**

If Statement: The if statement can be used to test conditions so that we can alter the flow of a program. Code:

```
#include<stdio.h>
  int main()
{
        int mark; char pass;
        scanf("%d",&mark);
        if (mark > 40) pass
        = "y";
                return 0;
}
```

If-Else Statement: The if statement first tests if a condition is true and then executes an instruction and the else is for when the result of the condition is false. Code:

```
#include<stdio.h> int
main()
{
        int mark; char pass;
        scanf("%d",&mark);
        if (mark > 40)
        {
                pass = "y";
                printf("You passed");
        }
        else
        {
                pass = "n"; printf("You
                failed");
        }
        return 0;
}
```

The Switch Statement: The switch statement is just like an if statement but it has many conditions and the commands for those conditions in only 1 statement. It is runs faster than an if statement. In a switch statement you first choose the variable to be tested and then you give each of the conditions and the commands for the conditions. You can also put in a default if none of the conditions are equal to the value of the variable. **For Loop, While Loop, Break and Continue**

Control structures are basically of three types –

- **Sequence statements**
- **Iterative statements**
- **Selection statements**

Sequence Statements : All the State in a program except the iterative & statements. They are generally the individual statements which performs the task of input, output, assignment declaration etc.

Iterative Statement are those repeated execution of a particular set of instructions desired number of times. These statements are generally called loops for their execution nature.

Types of Looping Statements:

Basically, the types of looping statements depends on the condition checking mode. Condition checking can be made in two ways as : Before loop and after loop. So, there are 2(two) types of looping statements.

- **Entry controlled loop** **Exit controlled loop**

2. Entry controlled loop :

In such type of loop, the test condition is checked first before the loop is executed.

Some common examples of this looping statements are :

- **while loop**
- **for loop**

2. Exit controlled loop :

In such type of loop, the loop is executed first. Then condition is checked after block of statements are executed. The loop executed atleat one time compulsarily.

Some common example of this looping statement is :

 do-while loop

In C language the iterative statements (loops) can be implemented in the three loops and they are

The For Loop

Syntax-

```
for (initialization; condition; incrementation)
{
    ----------------- body of loop -------------
}
```

For Loop will perform its execution until the condition remains satisfied. If the body of the loop consists of more than one statement then these statements are made compound by placing the open and closed curly brackets around the body of the loop. For loop is a count loop. The initialization condition and incrementation may be done in the same statement. For loop will not execute at least once also if the condition is false at the first time itself.

The While Loop

Syntax –

```
          Initialization;
      While (condition)
         {
         Body of loop;
                 Incrementation;
         }
```

In this loop, initialisation, condition and incrementation is done in the three different statements. This loops is count as well as event loop. In case of while loops the body of the loop will consist of more than one statements because each time one statement will be of incrementation. Hence the open and closed curly brackets are required.

Do-While Loop Statement:
The third loop statement available in C is do-while statement syntax :-

```
Initialization;
Do
{
Body of loop;
Increamentation;
}while(condition)
```

OPERATING SYSTEM

Operating System

An operating system is a program that acts as an interface between the software and the computer hardware.

- It is an integration set of specialized programs that are used to manage overall resources and operations of the computer.
- It is specialized software that controls and monitors the execution of all other programs that reside in the computer, including application programs and other system software.

Objectives of Operating System

- Making a computer system convenient to use in an efficient manner
- To hide the details of the hardware resources from the users
- To provide users a convenient interface to use the computer system.
- To act as an intermediary between the hardware and its users and making it easier for the users to access and use other resources.
- Manage the resources of a computer system.
- Keep track of who is using which resource, granting resource requests, according for resource using and mediating conflicting requests from different programs and users.
- The efficient and fair sharing of resources among users and programs

Characteristics of Operating System

- **Memory Management** -- It keeps tracks of primary memory i.e what part of it are in use by whom, what part are not in use etc. Allocates the memory when the process or program request it.
- **Processor Management** -- Allocate the processor(CPU) to a process. De-allocate processor when processor is no longer required.
- **Device Management** -- Keep tracks of all devices. This is also called I/O controller. Decides which process gets the device when and for how much time.
- **File Management** -- Allocates the resources. De-allocates the resources. Decides who gets the resources.
- **Security** -- By means of passwords & similar other techniques, preventing unauthorized access to programs & data.
- **Job accounting** -- Keeping track of time & resources used by various jobs and/or users.

- **Control over system performance** -- Recording delays between request for a service & from the system.
- **Interaction with the operators** -- The interaction may take place via the console of the computer in the form of instructions. Operating System acknowledges the same, do the corresponding action and inform the operation by a display screen.
- **Error-detecting aids** -- Production of dumps, traces, error messages and other debugging and error-detecting methods.
- **Coordination between other software and users** -- Coordination and assignment of compilers, interpreters, assemblers and other software to the various users of the computer systems.

WINDOWS 7

Windows 7 is an operating system produced by Microsoft for use on personal computers, including home and business desktops, laptops, net books, tablet PCs, and media center PCs. It was released to manufacturing on July 22, 2009, and became generally available retail worldwide on October 22, 2009, less than three years after the release of its predecessor, Windows Vista. Windows 7's server counterpart, Windows Server 2008 R2, was released at the same time. Windows 7 is succeeded by Windows 8.

Unlike Windows Vista's many new features, Windows 7 was an incremental upgrade designed to work with Vista-compatible applications and hardware. Presentations given by Microsoft in 2008 focused on multi-touch support, an updated Windows shell with a new taskbar, referred to internally as the *Superbar*, a home networking system called Home Group, and performance improvements. Some standard applications that have been included with prior releases of Microsoft Windows, including Windows Calendar, Windows Mail, Windows Movie Maker, and Windows Photo Gallery, are not included in Windows 7; most are instead offered separately at no charge as part of the Windows Essentials suite.

Install Windows 7

Many people have computers that come with Windows 7 so they may never have to install it. However, you may need to install Windows 7 if:

- You replaced your hard disk drive with a new hard disk drive that does not have Windows 7 installed.
- You are reinstalling Windows 7 on a computer because you want to clean off your hard drive and remove any unwanted programs, such as spyware. You purchased a computer without an operating system.

Pre-installation checklist

Before you begin the installation process, use this checklist to make sure that you are prepared:

- You have the Windows 7 CD.
- You have the product key available. The product key is located on your Windows 7 CD case and is required to install and activate Windows 7.
- Your computer hardware is set up. At a minimum, you should connect your keyboard, mouse, monitor, and CD drive. If available, you should connect your computer to a wired network.
- You have Windows 7 drivers available. Drivers are software that Windows 7 uses to communicate with your computer‘s hardware. If you do not have drivers available, Windows 7 may already include drivers for your hardware. If not, you should be able to download them from your hardware manufacturer‘s website after you set up Windows 7.
- If you are reinstalling Windows 7 on an existing computer, you need a backup copy of your files and settings. The installation process will delete all of your files. You can use the File and Settings Transfer Wizard to store your files and settings on removable media and then restore them after installation is complete.

Installation process

Installing Windows 7 can take up to two hours. To make the process more manageable, it has been broken up into several sections.

To Begin Installation:

1. Insert the Windows 7 CD into your computer and restart your computer.
2. If prompted to start from the CD, press Spacebar. If you miss the prompt (it only appears for a few seconds), restart your computer to try again.
3. Windows 7 Setup begins. During this portion of setup, your mouse will not work, so you must use the keyboard. On the Welcome to Setup page, press Enter.
4. On the Windows 7 Licensing Agreement page, read the licensing agreement. Press the Page Down key to scroll to the bottom of the agreement. Then press F8.
5. This page enables you to select the hard disk drive on which Windows 7 will be installed. Once you complete this step, all data on your hard disk drive will be removed and cannot be recovered. It is extremely important that you have a recent backup copy of your files before continuing. When you have a backup copy, press D, and then press L when prompted. This deletes your existing data.
6. Press Enter to select unpartitioned space, which appears by default.
7. Press Enter again to select Format the partition using the NTFS file system, which appears by default.
8. Windows 7 erases your hard disk drive using a process called formatting and then copies the setup files. You can leave your computer and return in 20 to 30 minutes.

To Continue the Installation

1. Windows 7 restarts and then continues with the installation process. From this point forward, you can use your mouse. Eventually, the Regional and Language

Options page appears. Click Next to accept the default settings. If you are multilingual or prefer a language other than English, you can change language settings after setup is complete.

2. On the Personalize Your Software page, type your name and your organization name. Some programs use this information to automatically fill in your name when required. Then, click next.
3. On the Your Product Key page, type your product key as it appears on your Windows 7 CD case. The product key is unique for every Windows 7 installation. Then, click Next.
4. On the Computer Name and Administrator Password page, in the Computer name box, type a name that uniquely identifies your computer in your house, such as FAMILYROOM or TOMS. You cannot use spaces or punctuation. If you connect your computer to a network, you will use this computer name to find shared files and printers. Type a strong password that you can remember in the Administrator password box, and then retype it in the Confirm password box. Write the password down and store it in a secure place. Click Next.
5. On the Date and Time Settings page, set your computer's clock. Then, click the Time Zone down arrow, and select your time zone. Click Next.
6. Windows 7 will spend about a minute configuring your computer. On the Networking Settings page, click next.
7. On the Workgroup or Computer Domain page, click Next.

To complete the installation

1. Windows 7 will spend 20 or 30 minutes configuring your computer and will automatically restart when finished. When the Display Settings dialog appears, click OK.
2. When the Monitor Settings dialog box appears, click OK.
3. The final stage of setup begins. On the Welcome to Microsoft Windows page, click Next.
4. On the Help protect your PC page, click Help protect my PC by turning on Automatic Updates now. Then, click Next.
5. Windows 7 will then check if you are connected to the Internet:
   - If you are connected to the Internet, select the choice that describes your network connection on the Will this computer connect to the Internet directly, or through a network page. If you're not sure, accept the default selection, and click Next.
   - If you use dial-up Internet access, or if Windows 7 cannot connect to the Internet, you can connect to the Internet after setup is complete. On the How will this computer connect to the Internet? page, click Skip.
6. Windows 7 Setup displays the Ready to activate Windows page. If you are connected to the Internet, click Yes, and then click Next. If you are not yet connected to the Internet, click No, click Next, and then skip to step 24. After setup is complete, Windows 7 will automatically remind you to activate and register your copy of Windows 7.
7. On the Ready to register with Microsoft page, click Yes, and then click Next.

8. On the Collecting Registration Information page, complete the form. Then, click Next.
9. On the Who will use this computer page, type the name of each person who will use the computer. You can use first names only, nicknames, or full names. Then click Next.
10. On the Thank you! Page, click Finish.

    Windows 7 setup is complete. You can log on by clicking your name on the logon screen. If you‘ve installed Windows 7 on a new computer or new hard disk drive, you can now use the File and Settings Transfer Wizard to copy your important data to your computer or hard disk drive.

MS-DOS

Short for **Microsoft Disk operating system**, **MS-DOS** is a non-graphical command line operating system derived from 86-DOS that was created for IBM compatible computers. MS-DOS originally written by Tim Peterson and introduced by Microsoft in August 1981 and was last updated in 1994 when MS-DOS 6.22 was released. Today, MS-DOS is no longer used; however, the command shell, more commonly known as the **Windows command line** is still used by many users.

Today, most computer users are only familiar with how to navigate Microsoft Windows using the mouse Unlike Windows, MS-DOS is a command-line and is navigated by using MS-DOS commands. For example, if you wanted to see all the files in a folder in Windows you would double-click the folder to open the folder in Windows Explorer. In MS-DOS, to view that same folder you would navigate to the folder using the CD command and then list the files in that folder using the dir command.

DOS Commands

MS-DOS has a relatively small number of commands, and an even smaller number of commonly used ones. Moreover, these commands are generally inflexible because, in contrast to Linux and other Unix-like operating systems, they are designed to accommodate few options or *arguments* (i.e., values that can be passed to the commands).

Some of the most common commands are as follows (corresponding commands on Unix-like operating systems are shown in parenthesis):

CD - changes the current directory (cd)
COPY - copies a file (cp)
DEL - deletes a file (rm)
DIR - lists directory contents (ls)
EDIT - starts an editor to create or edit plain text files (vi, vim, ed, Joe)
FORMAT - formats a disk to accept DOS files (mformat)
HELP - displays information about a command (man, info)
MKDIR - creates a new directory (mkdir)

RD - removes a directory (rmdir)
REN - renames a file (mv)
TYPE - displays contents of a file on the screen (more, cat)

Other DOS Command commonly used are:

Append

The append command can be used by programs to open files in another directory as if they were located in the current directory.

Attrib

The attrib command is used to change the attributes of a single file or a directory.

Break

The break command sets or clears extended CTRL+C checking. **Call**

The call command is used to run a script or batch program from within another script or batch program.

The call command has no effect outside of a script or batch file. In other words, running the call command at the DOS prompt will do nothing. **Chcp**

The chcp command displays or configures the active code page number. **Chdir**

The chdir command is used to display the drive letter and folder that you are currently in. Chdir can also be used to change the drive and/or directory that you want to work in.

Chkdsk

The chkdsk command, often referred to as *check disk*, is used to identify and correct certain hard drive errors.

Choice

The choice command is used within a script or batch program to provide a list of choices and return the value of that choice to the program.

Cls

The cls command clears the screen of all previously entered commands and other text.

Dir

The dir command is used to display a list of files and folders contained inside the folder that you are currently working in.

The dir command also displays other important information like the hard drive's serial number, the total number of files listed, their combined size, the total amount of free space left on the drive, and more.

DIR [drive:][path][filename] [/P] [/W] [/A[[:]attributes]] [/O[[:]sort order]] [/S] [/B] [/L] [/V]

[drive:][path][filename]	Specifies drive, directory, or files to list. (Could be enhanced file specification or multiple file specs)		
/P	Pauses after each screenful of information.		
/W	Uses wide list format.		
	attributes:		
	D		
	R	Read-only	Directories
/A	H	Hidden	files files
	A	Files ready for	archiving

	S System files - Prefix meaning not
/O	List by files in sorted order, sort order: N By name (alphabetic) S By size (smallest first) E By extension (alphabetic) D By date and time (earliest first) G Group directories first - Prefix to reverse order A By Last Access Date (earliest first)
/S	Displays files in specified directory and all subdirectories.
/B	Uses bare format (no heading information or summary).
/L	Uses lowercase.
/V	Verbose mode.

Computer Architecture

In computer science and engineering, computer architecture is the art that specifies the relations and parts of a computer system. .Computer architecture is different than the architecture of buildings, the latter is a form of visual arts while the former is part of computer sciences. In both instances (building and computer), a complete design has many details, and some details are implied by common practice.

For example, at a high level, computer architecture is concerned with how the central processing unit (CPU) acts and how it uses computer memory. Some fashionable (2011) computer architectures include cluster computing and Non-Uniform Memory Access.

Computer architects use computers to design new computers. Emulation software can run programs written in a proposed instruction set. While the design is very easy to change at this stage, compiler designers often collaborate with the architects, suggesting improvements in the instruction set. Modern emulators may measure time in clock cycles: estimate energy consumption in joules, and give realistic estimates of code size in bytes. These affect the convenience of the user, the life of a battery, and the size and expense of the computer's largest physical part: its memory. That is, they help to estimate the value of a computer design.

Computer organization

Computer organization helps optimize performance-based products. For example, software engineers need to know the processing ability of processors. They may need to optimize software in order to gain the most performance at the least expense. This can require quite detailed analysis of the computer organization. For example, in a multimedia decoder, the designers might need to arrange for most data to be processed in the fastest data path and the various components are assumed to be in place and task is to investigate the organisational structure to verify the computer parts operates.

Computer organization also helps plan the selection of a processor for a particular project. Multimedia projects may need very rapid data access, while supervisory software may need fast interrupts. Sometimes certain tasks need additional components as well. For example, a computer capable of virtualization needs virtual memory hardware so that the memory of different simulated computers can be kept separated. Computer organization and features also affect power consumption and processor cost.

Implementation

Once an instruction set and microarchitecture are described, a practical machine must be designed. This design process is called the *implementation*. Implementation is usually not considered architectural definition, but rather hardware design engineering. Implementation can be further broken down into several (not fully distinct) steps:

- Logic Implementation designs the blocks defined in the microarchitecture at (primarily) the register-transfer level and logic gate level.
- Circuit Implementation does transistor-level designs of basic elements (gates, multiplexers, latches etc.) as well as of some larger blocks (ALUs, caches etc.) that may be implemented at this level, or even (partly) at the physical level, for performance reasons.
- Physical Implementation draws physical circuits. The different circuit components are placed in a chip floorplan or on a board and the wires connecting them are routed.
- Design Validation tests the computer as a whole to see if it works in all situations and all timings. Once implementation starts, the first design validations are simulations using logic emulators. However, this is usually too slow to run realistic programs. So, after making corrections, prototypes are constructed using FieldProgrammable Gate-Arrays(FPGAs). Many hobby projects stop at this stage. The final step is to test prototype integrated circuits. Integrated circuits may require several redesigns to fix problems.

For CPUs, the entire implementation process is often called CPU design. Once an instruction set and microarchitecture ar described, a practical machine must be designed. This design process is called the implementation Implementation is usually not considered Implementation can be further broken down into several (not fully distinct) steps: Logic Implementation designs the blocks defined in the microarchitecture at (primarily) the register-transfer level and logic gate level.

Circuit implementation does transistor level designs of basic elements gates multiplexers, latches etc. as well as of some larger blocks arithmetic logic unit ALU s, caches etc. that may be implemented at this level, or even (partly) at the physical level, for performance reasons. Physical Implementation draws physical circuits.

The different circuit components are placed in a chip Floorplan (microelectronics) floorplan or on a board and the wires connecting them are routed. Design Validation tests the computer as a whole to see if it works in all situations and all timings. Once implementation starts, the first design validations are simulations using logic emulators. However, this is usually too slow to run realistic programs.

So, after making corrections, prototypes are constructed using Field-Programmable Gate-Arrays. Many hobby projects stop at this stage. The final step is to test prototype integrated circuits. Integrated circuits may require several redesigns to fix problems.

For Central processing unit CPU s, the entire implementation process is often called CPU design.

Design Goals

The exact form of a computer system depends on the constraints and goals. Computer architectures usually trade off standards, cost, memory capacity, latency (latency is the amount of time that it takes for information from one node to travel to the source) and throughput. Sometimes other considerations, such as features, size, weight, reliability, expandability and power consumption are factors.

The most common scheme carefully chooses the bottleneck that most reduces the computer's speed. Ideally, the cost is allocated proportionally to assure that the data rate is nearly the same for all parts of the computer, with the most costly part being the slowest. This is how skillful commercial integrators optimize personal computers such as smart cellphones.

Instruction set architecture (ISA) describes the processor in terms of what the assembly language programmer sees, i.e. the instructions and registers.

Organisation is concerned with the internal design of the processor, the design of the bus system and its interfaces, the design of memory and so on. Two machines may have the same ISA, but different organisations.

The organisation is implemented in hardware and in turn, two machines with the same organisation may have different hardware implementations, for example, a faster form of silicon technology may be used in the fabrication of the processor.

Instruction Set

One of the crucial features of any processor is its **instruction set**, i.e. the set of machine code instructions that the processor can carry out. Each processor has its own unique instruction set specifically designed to make best use of the capabilities of that processor. The actual number of instructions provided ranges from a few dozen for a simple 8-bit microprocessor to several hundred for a 32-bit processor. However, it should be pointed out that a large instruction set does not necessarily imply a more powerful processor.

Many modern processor designs are so called **RISC** (Reduced Instruction Set Computer) designs which use relatively small instruction sets, in contrast to so called **CISC** (Complex Instruction Set Computer) designs such as the VAX and machines based on the Intel 8086 and Motorola 68000 microprocessor families.

Classification of Instructions

The actual instructions provided by any processor can be broadly classified into the following groups:

- **Data movement** instructions: These allow the processor move data between registers and between memory and registers (e.g. 8086 mov, push, pop instructions). A _move‘ instruction and its variants is among the most frequently used instructions in an instructionset.

- **Transfer of control** instructions: These are concerned with branching for loops and conditional control structures as well as for handling subprograms (e.g.
8086 je, jg, jmp, call, ret instructions). These are also commonly used instructions.

- **Arithmetic/logical** instructions: These carry out the usual arithmetic and logical operations (e.g. 8086 cmp, add, sub, inc, and, or, xor instructions). Surprisingly, these are not frequently used instructions, and when used, it is often in conjunction with a conditional jump instruction rather than for general arithmetic purposes. Note that we have included the cmp instruction with the arithmetic/logical instructions because it actually behaves like a sub instruction except it does not modify its destination register.

- **Input/output** instructions: These are used for carrying out I/O (e.g. 8086 in, out instructions) but a very common form of I/O called memory mapped I/O uses _move‘ instructions for I/O.

- **Miscellaneous** instructions (e.g. 8086 int, sti, cti, hlt, nop) for handling interrupts and such activities. The hlt instruction halts the processor and the nop instruction does nothing at all! These instructions are again not that frequently used relative to data movement and transfer of control instructions.

Fixed and Variable length Instructions

Instructions are translated to machine code. In some architectures all machine code instructions are the same length i.e. fixed length. In other architectures, different instructions may be translated into variable lengths in machine code.

This is the situation with 8086 instructions which range from one byte to a maximum of 6 bytes in length. Such instructions are called variable length instructions and are commonly used on CISC machines.

The advantage of using such instructions, is that each instruction can use exactly the amount of space it requires, so that variable length instructions reduce the amount of memory space required for a program.

On the other hand, it is possible to have fixed length instructions, where as the name suggests, each instruction has the same length. Fixed length instructions are commonly used with RISC processors such as the PowerPC and Alpha processors.

Since each instruction occupies the same amount of space, every instruction must be long enough to specify a memory operand, even if the instruction does not use one. Hence, memory space is wasted by this form of instruction. The advantage of fixed length instructions, it is argued, is that they make the job of fetching and decoding instructions easier and more efficient, which means that they can be executed in less time than the corresponding variable length instructions.

Thus the comparison between fixed and variable length instructions comes down to the classic computing trade off of memory usage versus execution time.

In general, computer programs that execute very quickly tend to use larger amounts of storage, while programs to carry out the same tasks, that do not use so much storage, tend to take longer to execute.

Fetch-Execute Cycle

This is the fundamental operation of the processor. The CPU executes the instructions that it finds in the computers memory. In order to execute an instruction, the CPU must first fetch (transfer) the instruction from memory into one of its registers.This is a nontrivial task requiring several steps and is described later.The CPU then decodes the instruction, i.e. it decides which instruction has been fetched and finally it executes (carries out) the instruction.The CPU then repeats this procedure, i.e. it fetches an instruction, decodes and executes it. This process is repeated continuously and is known as the fetch-execute cycle.This cycle begins when the processor is switched on and continues until the CPU is halted (via a halt instruction, e.g. 8086 hlt instruction or the machine is switched off).Instead of looking at the details of a particular microprocessor's architecture at this point, we will use a simple hypothetical microprocessor to explain the basic concepts of computer architecture.

We call the machine SAM (Simple Architecture Machine). Figure 1 illustrates the major components of SAM. It is a 16-bit microprocessor with 4 general purpose registersr0 to r3, a program counter register PC, a stack pointer register SP and status register SR. The status register is made up of similar flags to the 8086 flags register, e.g. a zero flag, an overflow flag, a carry flag and so on.The fetch-execute cycle operates by first fetching an instruction. The program counter register PC always contains the address of the next instruction to be executed.
Let us assume that a particular program has been loaded into memory and is currently being executed. Program execution has reached a certain point (the move instruction is being executed) and the three instructions of the program are listed in Example 1.

To illustrate how the fetch execute cycle operates, we will trace the execution of these instructions. We assume that these instructions are stored in memory beginning at location 3000H. The instructions and their machine code equivalents (in hexadecimal) are listed below. We use hexadecimal instead of binary as it is easier to work with but you must remember that it is the binary form of the instructions that are actually stored in memory.

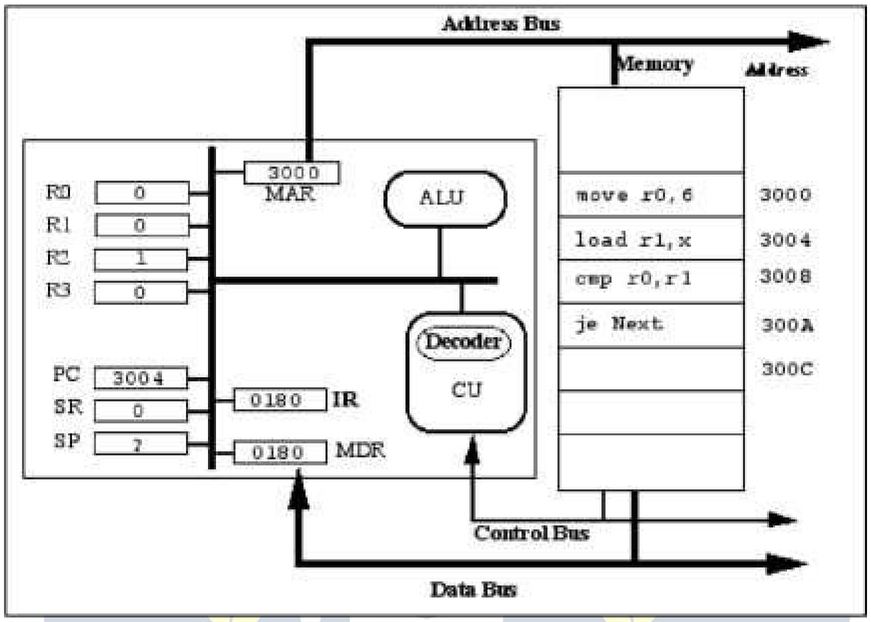


Fig. 1: SAM state after fetching move r0,6 (0180H in machine code)

Accessing Memory

In order to execute programs, a microprocessor fetches instructions from memory and executes them, fetching data from memory if it is required.

In Figure 1 we introduced two registers that we have not mentioned before namely the memory address register, MAR and the memory data register, MDR. There are a number of such CPU registers that do not appear in the programming model of a CPU. We shall refer to these registers as hidden CPU registers to distinguish them from the programming model registers R0 to R4, PC, SR and SP registers.

The MAR and MDR registers are used to communicate with memory (and other devices attached to the system bus).
In addition, Figure 1 shows the buses that allow the devices making up a SAM computer system communicate with each other.

The MAR register is used to store the address of the location in memory that is to be accessed for reading or writing.

When we retrieve information from memory we refer to the process as reading from memory.

When we store an item in memory, we refer to the process as writing to memory.

In either case, before we can access memory, we must specify the location we wish to access, i.e. the address of the location in memory. This address must be stored in theMAR register.

The MAR register is connected to memory via the address bus whose function is to transfer the address in the MAR register to memory. In this way the memory unit is informed as to which location is to be accessed.

The address bus is a uni-directional bus, i.e. information can only travel along it in a single direction, from the CPU to memory and other devices.

The MAR register is a 16-bit register like all the other SAM registers. This means that the maximum address it can contain is 216 - 1 (65,535) bytes, i.e. it can address up to 64Kb of memory.

The MDR register is used either to store information that is to be written to memory or to store information that has been read from memory. The MDR register is connected to memory via the data bus whose function is to transfer information, to or from memory and other devices.

The data bus is a bi-directional bus, i.e. information can travel along it, both, to and from the CPU.

The control bus plays a crucial role in I/O. It carries control signals specifying what operation is to be carried out and to synchronise the transfer of information.

For example, one line of the control bus is the read/write (R/W) line which used to specify whether a read or write operation is to be carried out.

Another line is the valid memory address (VMA) line which indicates that the address bus now carries a valid memory address. This tells the memory unit when to look at the address bus to find the address of the location to be accessed. A third line is the memory operation complete (MOC) line which signals that the read/write operation has now completed. We should note at this point, that the other devices attached to the computer, such as I/O and storage devices, usually communicate with the CPU in a similar fashion to that described for communicating with memory

Reading from Memory

The following steps are carried out by the SAM microprocessor to read an item from memory. The item may be an instruction or a data operand.

1. The address of the item in memory is stored in the MAR register.

2. This address is transferred to the address bus.

3. The VMA line and R/W line of the control bus are used to indicate to memory that there is a valid address on the address bus and that a read operation is to be carried out.

4. Memory responds by placing the contents of the desired address on the data bus.

5. Memory enables the MOC line to indicate that the memory operation is complete, i.e. the data bus contains the required data.

6. The information on the data bus is transferred to the MDR register.

7. The information is transferred from the MDR register to the specified CPU register.

Writing to Memory

This procedure is similar to that for reading from memory:

1. The address of the item in memory is stored in the MAR register.

2. This address is transferred to the address bus.

3. The item to be written to memory is transferred to the MDR register.

4. This information is transferred to the data bus.

5. The VMA line and R/W line of the control bus are used to indicate to memory that there is a valid address on the address bus and that a write operation is to be carried out.

6. Memory responds by placing the contents of the data bus in the desired memory location.
7. Memory uses the MOC line to indicate that the memory operation is complete, i.e. the data has been written to memory.

We can see from the above descriptions (which have been simplified!) that accessing memory or any device is quite complicated from an implementation viewpoint. So, when an instruction such as load r1, i

to load a register with the contents of a memory variable is to be executed, a lot of work has to be carried out.

Firstly the instruction must be fetched from RAM, then the value of i must be fetched from RAM and finally the transfer of the value of i to register r1 is carried out.

It is important to realise that every operation concerning memory involves either reading or writing memory.

Memory is a passive device. It can only store information. No processing can be carried out on information in memory. The information, stored in memory, must be transferred to a CPU register for processing and the result written back to memory.

So, for example, when an instruction such as the 8086 inc instruction is carried out to increment a memory variable (as in inc memvar), its execution involves both a memory read operation and a memory write operation.

Firstly, the value of memvar must be transferred to the CPU where it can be incremented by the ALU. This transfer is carried out via a memory read operation. Then, once this value has been incremented by the ALU, the new value of memvar must be written out to memvar's address in memory, via a memory write operation.

Encoding Instructions in Machine Code

Instructions are represented in machine code as binary numbers in same way as all other information is represented in a computer system. We noted earlier that assembly language instructions for most processors are broadly similar and have the form:

[label] operation [operand ..] [;comment]

The general form of a machine code instruction is illustrated in Figure 2 with the bits making up the instruction being grouped into opcode and operand fields.

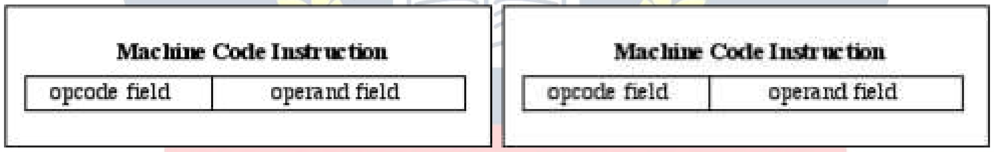


Figure 2: Machine code instruction format

The opcode field contains a binary code that specifies the operation to be carried out (e.g. add, jmp). Each operation has its own unique opcode. The operand field specifies the operand or operands that the operation is to be carried out on.. It should be emphasised that the instruction encoding for SAM is designed for illustration purposes. The aim is to keep it as simple as possible while remaining basically similar to the encoding of instructions on real processors. The reader is encouraged to look at ways the instructions could be more efficiently encoded.

Table 1 lists the opcodes of the commonly used SAM instructions in binary and hexadecimal.

Instruction	Binary Code	Hex Code
move	0000 0001	01
load	0000 0010	02

store	0000 0011	03
add	0000 1000	08
sub	0000 1001	09
cmp	0000 1111	0F
jmp	0001 1111	1F
je	0010 1111	2F

Table 1: SAM opcodes

The operand field of an instruction must be able to specify the registers, memory addresses or constants that the instruction is to operate on. SAM instructions have at most two operands. If there are two operands then one is always a register. If a memory address is specified (e.g. in the case of a memory variable or label) then the instruction is encoded using 32-bits. Since SAM has four general purpose registers we can represent them using 2-bit codes as follows:

00 for r0

01 for r1
10 for r2

11 for r3.

Thus, 4 bits are required to represent the two registers that may be used in an instruction. Bit numbers 0 and 1, represent the source register and bit numbers 2 and 3represent the destination register.

Visual Basic

Microsoft Visual Programming Language (VPL) is an application development environment designed on a graphical dataflow-based programming model. Rather than series of imperative commands sequentially executed, a dataflow program is more like a series of workers on an assembly line, who do their assigned task as the materials arrive. As a result VPL is well suited to programming a variety of concurrent or distributed processing scenarios.

VPL is targeted for beginner programmers with a basic understanding of concepts like variables and logic. However, VPL is not limited to novices. The programming language may appeal to more advanced programmers for rapid prototyping or code development. As a result, VPL may appeal to a wide audience of users including students, enthusiasts/hobbyists, as well as possibly web developers and professional programmers. In computing, a visual programming language (VPL) is any programming language that lets users create programs by manipulating program elements graphically rather than by specifying them textually. A VPL allows programming with visual expressions, spatial arrangements of text and graphic symbols used either as elements of syntax or secondary notation. For example, many VPLs (known as dataflow or diagrammatic programming) are based on the idea of "boxes and arrows", where boxes or other screen objects are treated as entities, connected by arrows, lines or arcs which represent relations.

VPLs may be further classified, according to the type and extent of visual expression used, into icon-based languages, form-based languages, and diagram languages. Visual programming environments provide graphical or iconic elements which can be manipulated by users in an interactive way according to some specific spatial grammar for program construction.

- A programming language that uses a visual representation (such as graphics, drawings, animation or icons, partially or completely)
- A visual language manipulates visual information or supports visual interaction, or allows programming with visual expressions [Golin 90]
- Any system where the user writes a program using two or more dimensions [Myers 90] A visual language is a set of spatial arrangements of text-graphic symbols with a semantic interpretation that is used in carrying out communication actions in the world
- VP Taxonomy by Burnett and Baker (1994)

Software paradigm, language level, application domain, visual extent (icon, form, diagram). Visual Basic and the entire Microsoft Visual (tm) family are not, despite their names, visual programming languages. They are textual languages which use a graphical gui builder to make programming decent interfaces easier on the programmer.

Elements of Visual Programming

- Language paradigm
- Batch or Interactive
- Visual representation vs. visual object
- Diagrams, icons or forms
- Number of dimensions
- Specific application domain

Basic Activities

The Microsoft Visual Programming Language (VPL) includes a set of basic activities that are used to help create a dataflow program. These blocks are typically used to connect between service blocks, but can also be connected together.

Activity

The activity block, sometimes referred to as a Custom Activity, allows you to create your own activities that can each have their own set of internal dataflow diagrams. You can use these to create diagrams that can be represented as single blocks in other diagrams and are used in the same way as the built-in activities. These custom activities can also be compiled into services that can be used with other services or in other VPL diagrams.

Calculate

The Calculate activity performs simple arithmetic or logical operations on the expression entered into the textbox. The expression can include numeric values, the value from the incoming message, its data members, or predetermined values provided by other services on your diagram.
For numeric data you can use:

+	add
-	subtract/minus
*	multiply
/	divide
%	mod (The modulus operation returns the remainder after a division)

The plus (+) operator can also be used to concatenate, combine, strings. This can also be used to combine text, string, and numeric data by using double quotes, e.g. the answer is + x/4.

For logical operators you can use:

&&	AND
||	OR
!	NOT

You can also use parenthesis to support precedence (the order of evaluation) of the expression entered.
Clicking in the textbox of the Calculate block displays a list including the value of the incoming message, any data members in the message, the state variables, as well as predefined values that may be provided by other services.

Variable

The Variable activity enables you to set or get the value of a variable.
To choose a variable select it from the drop-down list displayed by clicking on the downarrow button at the right-hand end of the text box.
If you have not defined any variables or wish to create a new variable, select Define Variables from the drop-down list, or select Variables from the Editmenu. This will display the Define Variables dialog box where you can add a variable and define its type. Variable data types include integers (int), double precision floating-point numbers (double), strings of characters (string), types of Lists, etc. See Data Types for the full list.
Variable names are case-sensitive. So when referencing a variable always be careful to use the same case. Names must also start with a letter and include only alphabetic or numeric characters. No punctuation characters are allowed except for the underscore (_).
Variable activities are simple constructs that support a GetValue connection, to get its value, as well as a SetValue connection, to set its value. When using the SetValue connection, the output connection of the activity also passes a variable on its outgoing connection.

Strings

- ➢ Variable Length
- ➢ Compare using standard comparators
- ➢ Maximum length is about 64Kb
- ➢ Minimum length is zero
- ➢ Allocated from VB ―String Space‖, so may run out of space even on systems with much memory.

Data Types

The Microsoft Visual Programming Language supports .NET C# style data types.

VPL Type	Description
bool	Boolean values: true, false
byte	8 bit unsigned integer (0 to 255)
sbyte	8 bit signed integer (-128 to 127)
char	character
decimal	fixed point decimal number (fixed precision number)
double	double precision (64-bit) floating point number (approx 14 significant digits)
float	single precision (32-bit) floating point number (approx 7 significant digits)

int	32 bit signed integer
uint	32 bit unsigned integer
long	64 bit signed integer
ulong	64 bit unsigned integer
short	16 bit signed integer (-32768 to 32767)
ushort	16 bit unsigned integer (0 to 65535)
string	character string (text)

Strategies in Visual Programming

Because VPEs employ visual ways of communicating about programs, the visual communication devices employed by a VPE can be viewed as a (limited) VPL. Hence, the strategies used by VPEs are a subset of those possible for VPLs. Because of this subset relationship, much of the remaining discussion of visual programming will focus primarily on VPLs.

To achieve these goals, there are four common strategies used in VPLs:

Concreteness: Concreteness is the opposite of abstractness, and means expressing some aspect of a program using particular instances. One example is allowing a programmer to specify some aspect of semantics on a specific object or value, and another example is having the system automatically display the effects of some portion of a program on a specific object or value.

Directness: Directness in the context of direct manipulation is usually described as ¯the feeling that one is directly manipulating the object‖. From a cognitive perspective, directness in computing means a small distance between a goal and the actions required of the user to achieve the goal. Given concreteness in a VPL, an example of directness would be allowing the programmer to manipulate a specific object or value directly to specify semantics rather than describing these semantics textually.

Explicitness: Some aspect of semantics is explicit in the environment if it is directly stated (textually or visually), without the requirement that the programmer infer it. An example of explicitness in a VPL would be for the system to explicitly depict dataflow relationships (program slice information) by drawing directed edges among related variables. Immediate Visual Feedback: In the context of visual programming, immediate visual feedback refers to automatic display of effects of program edits. Tanimoto has coined the term liveness, which categorizes the immediacy of semantic feedback that is automatically provided during the process of editing a program

Oracle

The Oracle Database (commonly referred to as Oracle RDBMS or simply as Oracle) is an object-relational database management system produced and marketed by Oracle Corporation.

Larry Ellison and his friends, former co-workers Bob Miner and Ed Oates, started the consultancy Software Development Laboratories (SDL) in 1977. SDL developed the original version of the Oracle software. The name Oracle comes from the code-name of a CIA-funded project Ellison had worked on while previously employed by Ampex.

Physical and Logical Structure

An Oracle database system—identified by an alphanumeric system identifier or SID—comprises at least one instance of the application, along with data storage. An instance—identified persistently by an instantiation number (or activation id: SYS.V_$DATABASE.ACTIVATION#)—comprises a set of operating-system processes and memory-structures that interact with the storage. (Typical processes include PMON (the process monitor) and SMON (the system monitor).) Oracle documentation can refer to an active database instance as a "shared memory realm".

Users of Oracle databases refer to the server-side memory-structure as the SGA (System Global Area). The SGA typically holds cache information such as databuffers, SQL commands, and user information.

In addition to storage, the database consists of online redo logs (or logs), which hold transactional history. Processes can in turn archive the online redo logs into archive logs (offline redo logs), which provide the basis (if necessary) for data recovery and for the physical-standby forms of data replication using Oracle Data Guard.

If the Oracle database administrator has implemented Oracle RAC (Real Application Clusters), then multiple instances, usually on different servers, attach to a central storage array. This scenario offers advantages such as better performance, scalability and redundancy. However, support becomes more complex, and many sites do not use RAC. In version 10g, grid computing introduced shared resources where an instance can use (for example) CPU resources from another node (computer) in the grid.

The Oracle DBMS can store and execute stored procedures and functions within itself. PL/SQL (Oracle Corporation's proprietary procedural extension to SQL), or the object-oriented language Java can invoke such code objects and/or provide the programming structures for writing them.

Storage

The Oracle RDBMS stores data logically in the form of tablespaces and physically in the form of data files ("datafiles"). Tablespaces can contain various types of memory segments, such as Data Segments, Index Segments, etc. Segments in turn comprise one or more extents. Extents comprise groups of contiguous data blocks. Data blocks form the basic units of data storage.

Newer versions of the database can also include a partitioning feature: this allows the partitioning of tables based on different set of keys. Specific partitions can then be easily added or dropped to help manage large data sets. Partitioning is useful for very large tables. By splitting a large table‘s rows across multiple smaller partitions, you accomplish several important goals:

- Backup and recovery operations may perform better. Because the partitions are smaller than the partitioned table, you may have more options for backing up and recovering the partitions than you would have for a single large table.
- The table may be easier to manage. Because the partitioned table‘s data is stored in multiple parts, it may be easier to load and delete data in the partitions than in the large table.
- The performance of queries against the tables may improve because Oracle may have to search only one partition (one part of the table) instead of the entire table to resolve a query.

Oracle database management tracks its computer data storage with the help of information stored in the SYSTEM tablespace. The SYSTEM tablespace contains the data dictionary—and often (by default) indexes and clusters.

A data dictionary consists of a special collection of tables that contains information about all user-objects in the database. Since version 8i, the Oracle RDBMS also supports "locally managed" tablespaces which can store space management information in bitmaps in their own headers rather than in the SYSTEM tablespace (as happens with the default "dictionary-managed" tablespaces).

Version 10g and later introduced the SYSAUX tablespace which contains some of the tables formerly in the SYSTEM tablespace.

Database Schema

Most Oracle database installations traditionally came with a default schema called SCOTT. After the installation process has set up the sample tables, the user can log into the database with the username scott and the password tiger. The name of the SCOTT schema originated with Bruce Scott, one of the first employees at Oracle (then Software Development Laboratories), who had a cat named Tiger.

Oracle Corporation has de-emphasized the use of the SCOTT schema, as it uses few of the features of the more recent releases of Oracle. Most recent examples supplied by Oracle Corporation reference the default HR or OE schemas.

Other default schemas include:

- SYS (essential core database structures and utilities)
- SYSTEM (additional core database structures and utilities, and privileged account)
- OUTLN (utilized to store metadata for stored outlines for stable query-optimizer execution plans.
- BI, IX, HR, OE, PM, and SH (expanded sample schemas containing more data and structures than the older SCOTT schema).

System Global Area

Each Oracle instance uses a System Global Area or SGA—a shared-memory area—to store its data and control-information. Each Oracle instance allocates itself an SGA when it starts and de-allocates it at shut-down time. The information in the SGA consists of the following elements, each of which has a fixed size, established at instance startup:

- the redo log buffer: this stores redo entries—a log of changes made to the database. The instance writes redo log buffers to the redo log as quickly and efficiently as possible. The redo log aids in instance recovery in the event of a system failure.
- the shared pool: this area of the SGA stores shared-memory structures such as shared SQL areas in the library cache and internal information in the data dictionary. An insufficient amount of memory allocated to the shared pool can cause performance degradation.
- the Large pool Optional area that provides large memory allocations for certain large processes, such as Oracle backup and recovery operations, and I/O server processes
- Database buffer cache: Caches blocks of data retrieved from the database
- KEEP buffer pool: A specialized type of database buffer cache that is tuned to retain blocks of data in memory for long periods of time
- RECYCLE buffer pool: A specialized type of database buffer cache that is tuned to recycle or remove block from memory quickly
- nK buffer cache: One of several specialized database buffer caches designed to hold block sizes different than the default database block size
- Java pool:Used for all session-specific Java code and data in the Java Virtual Machine (JVM)

- Streams pool: Used by Oracle Streams to store information required by capture and apply

When you start the instance by using Enterprise Manager or SQL*Plus, the amount of memory allocated for the SGA is displayed.

Concurrency and locking

Oracle databases control simultaneous access to data resources with locks (alternatively documented as "enqueues"). The databases also utilize "latches" - low-level serialization mechanisms to protect shared data structures in the System Global Area.

Configuration

Database administrators control many of the tunable variations in an Oracle instance by means of values in a parameter file. This file in its ASCII default form ("pfile") normally has a name of the format init<SID-name>.ora. The default binary equivalent server parameter file ("spfile") (dynamically reconfigurable to some extent) defaults to the format spfile<SID-name>.ora. Within an SQL-based environment, the views V$PARAMETER and V$SPPARAMETER give access to reading parameter values.

Oracle Database Features

Apart from the clearly defined database options, Oracle databases may include many semi-autonomous software sub-systems, which Oracle Corporation sometimes refers to as "features" in a sense subtly different from the normal usage of the word. For example, Oracle Data Guard counts officially as a "feature", but the command-stack within SQL*Plus, though a usability feature, does not appear in the list of "features" in Oracle's list. Such "features" may include (for example):

- Active Session History (ASH), the collection of data for immediate monitoring of very recent database activity.
- Automatic Workload Repository (AWR), providing monitoring services to Oracle database installations from Oracle version 10. Prior to the release of Oracle version 10, the Statspack facility provided similar functionality.
- Clusterware
- Data Aggregation and Consolidation
- Data Guard for high availability
- Generic Connectivity for connecting to non-Oracle systems.
- Data Pump utilities, which aid in importing and exporting data and metadata between databases
- Database Resource Manager (DRM), which controls the use of computing resources.
- Fast-start parallel rollback
- Fine-grained auditing (FGA) (in Oracle Enterprise Edition) supplements standard security-auditing features
- Flashback for selective data recovery and reconstruction
- iSQL*Plus, a web-browser-based graphical user interface (GUI) for Oracle database data-manipulation (compare SQL*Plus)
- Oracle Data Access Components (ODAC), tools which consist of:

- Oracle Data Provider for .NET (ODP.NET)
- Oracle Developer Tools (ODT) for Visual Studio
- Oracle Providers for ASP.NET
- Oracle Database Extensions for .NET
- Oracle Provider for OLE DB
- Oracle Objects for OLE
- Oracle Services for Microsoft Transaction Server

➢ Oracle-managed files (OMF) -- a feature allowing automated naming, creation and deletion of datafiles at the operating-system level.

➢ Recovery Manager (rman) for database backup, restoration and recovery

➢ SQL*Plus, a program that allows users to interact with Oracle database(s) via SQL and PL/SQL commands on a command-line. Compare iSQL*Plus.

➢ Universal Connection Pool (UCP), a connection pool based on Java and supporting JDBC, LDAP, and JCA

➢ Virtual Private Database (VPD), an implementation of fine-grained access control.

- This list is incomplete; you can help by expanding it.

Data Structure and Algorithms

Introduction

In computer science, a data structure is a particular way of storing and organizing data in a computer so that it can be used efficiently. Different kinds of data structures are suited to different kinds of applications, and some are highly specialized to specific tasks. For example, B-trees are particularly well-suited for implementation of databases, while compiler implementations usually use hash tables to look up identifiers . Data structures provide a means to manage large amounts of data efficiently, such as large databases and internet indexing services. Usually, efficient data structures are a key to designing efficient algorithms. Some formal design methods and programming languages emphasize data structures, rather than algorithms, as the key organizing factor in software design. Storing and retrieving can be carried out on data stored in both main memory and in secondary memory**.**

there are only three important ideas which must be mastered to write interesting programs.

- Iteration - Do, While, Repeat, If
- Data Representation - variables and pointers
- Subprograms and Recursion - modular design and abstraction

At this point, you have mastered about 1.5 of these 3.It is the purpose of Computer Science II to finish the job. Data types vs. Data Structures

A data type is a well-defined collection of data with a well-defined set of operations on it.

A data structure is an actual implementation of a particular abstract data type.

Design and Analysis of Algorithms

There are many steps involved in writing a computer program to solve a given problem. The steps go from problem formulation and specification, to design of the solution, to implementation , testing and documentation, and finally to evaluation of the solution. This section outlines our approach to these steps. Subsequent we will discuss the algorithms and data structures that are the building blocks of most computer programs.

Algorithms

Once we have a suitable mathematical model for our problem, we can attempt to find a solution in terms of that model. Our initial goal is to find a solution in the form of an algorithm, which is a finite sequence of instructions, each of which has a clear meaning and can be performed with a finite amount of effort in a finite length of time.

Linked List

Search, Insert, Delete

There are three fundamental operations we need for any database:

- Insert: add a new record at a given point
- Delete: remove an old record
- Search: find a record with a given key

We will see a wide variety of different implementation of these operations over the course of the semester.How would you implement these using an array?With linked lists, we can creating arbitrarily large structures, and never have to move any items.Most of these operations should be pretty simple now that you understand pointers!

Searching in a Linked List

```
Procedure Search(head:pointer, key:item):pointer;
Var
  p:pointer;
found:boolean;
Begin
```

```
found:=false;
p:=head;
   While (p # NIL) AND (not found) Do
    Begin
      If (p^.info = key) then
found = true;        Else
         p = p^.next;
End;    return p;
END;
```

Search performs better when the item is near the front of the list than the back.

Insertion into a Linked List

The easiest way to insert a new node p into a linked list is to insert it at the front of the list:

```
p^.next = front; front
= p;
```

To maintain lists in sorted order, however, we will want to insert a node between the two appropriate nodes. This means that as we traverse the list we must keep pointers to both the current node and the previous node.

```
MODULE Intlist;                    (*16.07.94. RM, LB*)
(* Implementation of sorted integer lists. *)

  REVEAL                           (*reveal inner structure of T*)
    T = BRANDED REF RECORD
         key: INTEGER;             (*key value*)
         next: T := NIL;          (*pointer to next element*)
       END; (*T*)

  PROCEDURE Create(): T =
  (* returns a new, empty list *)
  BEGIN
    RETURN NIL; (*creation is trivial; empty list is NIL*)
  END Create;

  PROCEDURE Insert(VAR list: T; value:INTEGER) =
  (* inserts new element in list and maintains order *)
  VAR
    current, previous: T;
    new: T := NEW(T, key:= value);   (*create new element*)
  BEGIN
    IF list = NIL THEN list:= new    (*first element*)
ELSIF value < list.key THEN      (*insert at beginning*)
new.next:= list; list:= new;
```

```
   ELSE                          (*find position for insertion*)
current:= list;        previous:= current;
     WHILE (current # NIL) AND (current.key <= value) DO
previous:= current;          (*previous hobbles after*)        current:=
current.next;
      END;
      (*after  the  loop  previous  points  to  the  insertion  point*)
new.next:= current;
          (*current   =   NIL   if   insertion   point   is   the   end*)
previous.next:= new;           (*insert new element*)
    END; (*IF list = NIL*)
  END Insert;
```

Make sure you understand where these cases come from and can verify why all of them work correct.

Deletion of a Node

To delete a node from a singly linked-list, we must have pointers to both the node-to-die and the previous node, so we can reconnect the rest of the list.

```
  PROCEDURE Remove(VAR list: T; value:INTEGER; VAR found: BOOLEAN) =
  (* deletes  (first) element with value from sorted list,      or
returns false in found if the element was not found *)
VAR
    current, previous: T;
  BEGIN
    IF list = NIL THEN found:= FALSE
    ELSE                   (*start search*)     current:=
list; previous:= current;
     WHILE (current # NIL) AND (current.key # value) DO
previous:= current;          (*previous hobbles after*)       current:=
current.next;
      END;
      (*holds: current = NIL or current.key = value, but not both*)
      IF current = NIL THEN
        found:= FALSE           (*value not found*)      ELSE
        found:= TRUE;                  (*value found*)
IF current = list THEN
         list:=  current.next              (*element  found  at  beginning*)
ELSE
         previous.next:= current.next
        END;
      END; (*IF current = NIL*)
    END; (*IF list = NIL*)
  END Remove;
```

Stacks and Queues

The first data structures we will study this semester will be lists which have the property that the order in which the items are used is determined by the order they arrive.

- Stacks are data structures which maintain the order of last-in, first-out
- Queues are data structures which maintain the order of first-in, first-out

Queues might seem fairer, which is why lines at stores are organized as queues instead of stacks, but both have important applications in programs as a data structure.

Operations on Stacks

The terminology associated with stacks comes from the spring loaded plate containers common in dining halls.

When a new plate is washed it is pushed on the stack.

When someone is hungry, a clean plate is popped off the stack.

Abstract Operations on a Stack

- Push(x,s) and Pop(x,s) - Stack s, item x. Note that there is no search operation.
- Initialize(s), Full(s), Empty(s), - The latter two are Boolean queries.

Defining these abstract operations lets us build a stack module to use and reuse without knowing the details of the implementation.

The easiest implementation uses an array with an index variable to represent the top of the stack.

An alternative implementation, using linked lists is sometimes better, for it can't ever overflow. Note that we can change the implementations without the rest of the program knowing!

Declarations for a stack

```
INTERFACE Stack;                (*14.07.94 RM, LB*)
(* Stack of integer elements *)

  TYPE ET = INTEGER;            (*element type*)

  PROCEDURE Push(elem : ET);      (*adds element to top of stack*)
  PROCEDURE Pop(): ET;            (*removes and returns top element*)
  PROCEDURE Empty(): BOOLEAN;     (*returns true if stack is empty*) PROCEDURE
Full(): BOOLEAN;      (*returns true if stack is full*)

END Stack.
```

A stack is an appropriate data structure for this task since the plates don't care about when they are used!

Queues

Queues are more difficult to implement than stacks, because action happens at both ends. The *easiest* implementation uses an array, adds elements at one end, and *moves* all elements when something is taken off the queue. It is very wasteful moving all the elements on each DEQUEUE. Can we do better?

More Efficient Queues

Suppose that we maintaining pointers to the first (head) and last (tail) elements in the array/queue? Note that there is no reason to explicitly clear previously unused cells.

Now both *ENQUEUE* and *DEQUEUE* are fast, but they are wasteful of space. We need a array bigger than the total number of *ENQUEUEs*, instead of the maximum number of items stored at a particular time.

Circular Queues

Circular queues let us reuse empty space!Note that the pointer to the front of the list is now *behind* the back pointer!When the queue is full, the two pointers point to neighboring elements.

Other Queues

Double-ended queues - These are data structures which support both push and pop and enqueue / dequeue operations. *Priority Queues(heaps)* - Supports insertions and ``remove minimum'' operations which useful in simulations to maintain a queue of time events.

Pointers and Dynamic Memory Allocation

Although arrays are good things, we cannot adjust the size of them in the middle of the program.If our array is too small - our program will fail for large data.If our array is too big - we waste a lot of space, again restricting what we can do.The right solution is to build the data structure from small pieces, and add a new piece whenever we need to make it larger.

Pointers are the connections which hold these pieces together!Pointers in Real LifeIn many ways, telephone numbers serve as pointers in today's society.

- To contact someone, you do not have to carry them with you at all times. All you need is their number.
- Many different people can all have your number simultaneously. All you need do is copy the pointer.
- More complicated structures can be built by combining pointers. For example, phone trees or directory information.

Addresses are a more physically correct analogy for pointers, since they really are memory addresses.

Linked Data Structures

All the dynamic data structures we will build have certain shared properties.

- We need a pointer to the entire object so we can find it. Note that this is a pointer, not a cell.
- Each cell contains one or more data fields, which is what we want to store.
- Each cell contains a pointer field to at least one ``next'' cell. Thus much of the space used in linked data structures is not data!
- We must be able to detect the end of the data structure. This is why we need the NIL pointer.

Dynamic Allocation

To get dynamic allocation, use new:

```
p := New(ptype);
```

New(ptype) allocates enough space to store exactly one object of the type ptype. Further, it returns a pointer to this empty cell. Before a new or otherwise explicit initialization, a pointer variable has an arbitrary value which points to trouble!

Warning - initialize all pointers before use. Since you cannot initialize them to explicit constants, your only choices are

- NIL - meaning explicitly nothing.
- New(ptype) - a fresh chunk of memory.
- assignment to some previously initialized pointer of the same type.

Garbage Collection

The system is constantly keeping watch on the dynamic memory which it has allocated, making sure that *something* is still pointing to it. If not, there is no way for you to get access to it, so the space might as well be recycled. The *garbage collector* automatically frees up the memory which has nothing pointing to it.

It frees you from having to worry about explicitly freeing memory, at the cost of leaving certain structures which it can't figure out are really garbage, such as a circular list.

Recursion

Recursion is a wonderful, powerful way to solve problems.Elegant recursive procedures seem to work by magic, but the magic is same reason *mathematical induction* works! Recursion not only made a complicated problem understandable, it made it easy to understand.

Gray codes

We saw how to generate subsets recursively. Now let us generate them in an interesting order.

All subsets of $\{1,2,\ldots,n\}$ can be represented as binary strings of length n, where bit i tells whether i is in the subset or not.

Obviously, all subsets must differ in at least one element, or else they would be identical. An order where they differ by exactly one from each other is called a *Gray code*.

For n=1, {},{1}.

For n=2, {},{1},{1,2},{2}.

For n=3, {},{1},{1,2},{2},{2,3},{1,2,3},{1,3},{3}

Recursive construction algorithm: Build a Gray Code of $\{1,..,n-1\}$, make a reverse copy of it, append n to each subset in the reverse copy, and stick the two together!

Formulating Recursive Programs

Think about the *base cases*, the small cases where the problem is simple enough to solve.

Think about the *general case*, which you can solve if you can solve the smaller cases.

Unfortunately, many of the simple examples of recursion are equally well done by iteration, making students suspicious.

Further, many of these classic problems have hidden costs which make recursion *seem* expensive, but don't be fooled!

Factorials

```
PROCEDURE Factorial (n: CARDINAL): CARDINAL =
  BEGIN
    IF n = 0 THEN
      RETURN 1                    (* trivial case *)
    ELSE
      RETURN n * Factorial(n-1)     (* recursive branch *)
    END (* IF*)
  END Factorial;
```

Be sure you *understand* how the parameter passing mechanism works.

Recursive Descent Compilation

Compilers do two useful things

 They identify whether a program is legal in the language.
They translate it into assembly language.

To do either, we need a precise description of the language, a BNF grammar which gives the syntax. A grammar for Modula-3 is given throughout your text.The language definition can be recursive!!Our compiler will follow the grammar to break the program into smaller and smaller pieces.When the pieces get small enough, we can spit out the appropriate chunk of assembly code.

To avoid getting into infinite loops, we place our trust in the fellow who wrote the grammar. Proper design can ensure that there are no such troubles.

Introduction to Sorting

Sorting is, without doubt, the most fundamental algorithmic problem

1. Supposedly, 25% of all CPU cycles are spent sorting
2. Sorting is fundamental to most other algorithmic problems, for example binary search.
3. Many different approaches lead to useful sorting algorithms, and these ideas can be used to solve many other problems.

Issues in Sorting

Increasing or Decreasing Order? - The same algorithm can be used by both all we need do is change $\leq$ to $\geq$ in the comparison function as we desire.

What about equal keys? - Does the order matter or not? Maybe we need to sort on secondary keys, or leave in the same order as the original permutations.

What about non-numerical data? - Alphabetizing is sorting text strings, and libraries have very complicated rules concerning punctuation, etc. Is *Brown-Williams* before or after *Brown America* before or after *Brown, John*?

We can ignore all three of these issues by assuming a *comparison function* which depends on the application. *Compare (a,b)* should return ``<'', ``>'', or "=".

Applications of Sorting

One reason why sorting is so important is that once a set of items is sorted, many other problems become easy. SearchingBinary search lets you test whether an item is in a dictionary in $O(\lg n)$ time.Speeding up searching is perhaps the most important application of sorting.

There are several different ideas which lead to sorting algorithms:

- Insertion - putting an element in the appropriate place in a sorted list yields a larger sorted list.
- Exchange - rearrange pairs of elements which are out of order, until no such pairs remain.
- Selection - extract the largest element form the list, remove it, and repeat.
- Distribution - separate into piles based on the first letter, then sort each pile. Merging - Two sorted lists can be easily combined to form a sorted list.

Selection Sort

In my opinion, the most natural and easiest sorting algorithm is selection sort, where we repeatedly find the smallest element, move it to the front, then repeat...

```
*5   7   3   2   8
 2 * 7   3   5   8
 2   3 * 7   5   8
 2   3   5 * 7   8
 2   3   5   7 * 8
```

If elements are in an array, swap the first with the smallest element- thus only one array is necessary. If elements are in a linked list, we must keep two lists, one sorted and one unsorted, and always add the new element to the back of the sorted list.

Selection Sort Implementation

```
MODULE SimpleSort EXPORTS Main; (*1.12.94. LB*)
(* Sorting and text-array by selecting the smallest element *)

  TYPE
    Array = ARRAY [1..N] OF TEXT;
  VAR
    a: Array;          (*the array in which to search*)
x: TEXT;                  (*auxiliary variable*)
    last,                    (*last valid index *)   min:
INTEGER;               (* current minimum*)

BEGIN

...

  FOR i:= FIRST(a) TO last - 1 DO
    min:= i;                         (*index of smallest element*)
    FOR j:= i + 1 TO last DO
      IF Text.Compare(a[j], a[min]) = -1 THEN (*IF a[i] < a[min]*)
min:= j
      END;
    END; (*FOR j*)
    x:= a[min];                    (* swap a[i] and a[min] *)
a[min]:= a[i];     a[i]:= x;
  END; (*FOR i*)
   ...
END SimpleSort.
```

Insertion Sort

In insertion sort, we repeatedly add elements to a sorted subset of our data, inserting the next element in order. In inserting the element in the sorted section, we might have to move many elements to make room for it.

Trees

``I think that I shall never see a poem as lovely as a tree.
Poems are wrote by fools like me, but only G-d can make a tree.'' - Joyce Kilmer

We have seen many data structures which allow fast search, but not fast, flexible update.

Sorted Tables - $O(\log n)$ search, O(n) insertion, O(n) deletion.

Hash Tables - The number of insertions are essentially bounded by the table size, which must be specified in advance. Worst case O(n) search.

Binary trees will enable us to search, insert, and delete fast, without predefining the size of our data structure!

How can we get this flexibility?

The only data structure we have seen which allows fast insertion/ deletion is the linked list, with updates in O(1) time but search in O(n) time. To get $O(\log n)$ search time, we used binary search, meaning we always had a choice of two next elements to look at. To combine these ideas, we want a ``linked list'' with two pointers per node! This is the basic idea behind search trees!

Rooted Trees

We can use a recursive definition to specify what we mean by a ``rooted tree''.A rooted tree is either (1) empty, or (2) consists of a node called the root, together with two rooted trees called the left subtree and right subtree of the root.A binary tree is a rooted tree where each node has at most two descendants, the left child and the right child.

A binary tree can be implemented where each node has left and right pointer fields, an (optional) parent pointer, and a data field.

Rooted trees in Real Life

Rooted trees can be used to model corporate heirarchies and family trees.Note the inherently recursive structure of rooted trees. Deleting the root gives rise to a certain number of smaller subtrees.In a rooted tree, the order among ``brother'' nodes matters. Thus left is different from right. The five distinct binary trees with five nodes:

Binary Search Trees

A binary search tree is a binary tree where each node contains a key such that:

- All keys in the left subtree precede the key in the root.
- All keys in the right subtree succeed the key in the root.
- The left and right subtrees of the root are again binary search trees.

Left: A binary search tree. Right: A heap but not a binary search tree.

For any binary tree on n nodes, and any set of n keys, there is exactly one labeling to make it a binary search tree!!

Binary Tree Search

Searching a binary tree is almost like binary search! The difference is that instead of searching an array and defining the middle element ourselves, we just follow the appropriate pointer! The type declaration is simply a linked list node with another pointer. Left and right pointers are identical types.

```
TYPE
       T = BRANDED REF RECORD
        key:  ElemT;
left, right: T := NIL;
END; (*T*)
```

Dictionary search operations are easy in binary trees. The algorithm works because both the left and right subtrees of a binary search tree are binary search trees - recursive structure, recursive algorithm.