C Programming

Structured Programming

- Structured programming calls for a disciplined way of programming
- Minimize the use of the goto statement
- Write readable programs
- Write modular programs

The C Programming Language

- In 1972, Dennis Ritchie developed C as a powerful, general purpose structured programming language
- Now C programmers say
- C is not just a language, C is an attitude, C is a culture, C is a way of living

Structure of a C program

```
main(){
}
```

- The program written in C is first given to a preprocessor that expands the program into the real source code before giving it to the C compiler
- A C program may contain statements and preprocessor directives. The preprocessor directive will expand the program based on these directives
- All statement should end with a semi colon (;)
- A statement exceeding 79 columns is difficult to handle. A statement in C can be written in several lines. White spaces can be included between the keywords to improve readability

The C Library

- A C program can be stored in different files for easy management
- It is usual to store commonly used sub programs in a separate file and include them into the program in which you want to use them. Such files are called header files and they usually have an extension h and these sub programs are called function
- We use the #include preprocessor directive to include a file in our program

General Form

```
#include "filename"
#include <filename>
```

Example

```
#include "myfile.h"
#include <myfile.h>
```

- First form will look for the file in the current directory and then in the specified directories whereas the second form will look only in the specified directories

- A lot of commonly used functions written in C are readily available as libraries. They are organized into different header files like stdio.h, math.h, graphics.h etc. To use a library function, the corresponding header file is to be included in the program by the preprocessor

Comments in a C Program

- Including comments in a program is very important to make it more readable and maintainable. Any thing that starts with a /* and ends with a */ is a comment in C

Example

```
main(){
/*This is a
comment in
C*/
}
```

Data Types

- Data is stored in a computer as 1s and 0s. Data types will abstract this fact and specify a set of operations that can be done on these 1s and 0s. Some old languages were very problematic and dangerous because of lack of data types

Data Type in C	Length in bytes
Char	1
Int	2
Float	4
unsigned char	1
short int	2
long int	4
unsigned short int	2
unsigned long int	4
Double	8
long double	10

sizeof Operator

- The length of a data type in terms of number of bytes can be found using the sizeof operator

General Form

sizeof(Data Type)

Example

sizeof(int) will give the result as 2

Variables

- Data is stored in the main memory of the computer which is identified using a unique number called address
- Variable names are temporary names given to these locations for making them easy to use. The value of a variable can change

Constant

- Constant is something that cannot change. 10 is an integer constant, 3.96 is a float constant and '*' is a char constant
- 10L or 10l is a long constant
- -10U or -10u is an unsigned int constant

Declaration of variables

- All variables should be declared before use

General Form

DataType VariableName;

Example

```
int i;
float a;
```

- A number of variables can be declared in a single statement using comma

General Form

DataType VariableName1, VariableName2, VariableName3, …;

Example

```
int i,j,k;
float a,b;
```

- These variables are not automatically initialized. So they contain some garbage values
- The operator sizeof can be used with a variable also

General Form

sizeof(variable)

Example

sizeof(i)

Address Operator

- The address of a variable can be obtained using the address operator &

General Form

&VariableName

Example

&a

Storage Class

- To fully define a variable we should know its storage class also. Storage class will tell us
- Where the variable is stored? What is the initial value? What is the scope? How long would the variable exist?

Register Storage Class

- Some variables can be stored in the CPU register for faster access

Example

```
register int i;
```

- i will be stored in the CPU register, if possible
- It may not be able to store a float or double in a register as registers are usually only 2 bytes long
- If a variable can be changed by a background routine it should be loaded from the memory every time with out taking the advantage of having a copy in the register. Such variables are called volatile variables. This can be specified by using the keyword volatile

General Form

```
volatile declaration;
```

Example

```
volatile int j;
```

Initialization

- Along with declaring a variable, it can also be initialized

General Form

Data Type VariableName1=Constant1/Variable1, VariableName2=Constant2/Variable2, ;

Example

```
int i=10;
float x=3.96;
int a, b=10, c=b;
char m='A';
```

Assignment

- Assignment operator (=) can be used to assign a value to a variable

General Form

```
Variable=Variable/Constant;
```

Example

```
i=25;
a=23.45;
c='X';
```

- Variables, constants and operators can be used together to form expressions. So i=25 is an assignment expression
- Some times there can be more that one operator in an expression

Example

```
a=b=c=25;
```

- Such an expression is evaluated based on two things - Hierarchy of the operators and Direction of association of the operators. In this example, all operators are assignment operators having the same hierarchy. Assignment operator associates from right to left. So first 25 is assigned to c, then to b and then to a
- We can use parenthesis to change the normal hierarchy of operators

Overflow and Underflow

- Any data type can hold only a particular range of values. int can hold values from -32,768 to +32, 767. Trying to assign a value above this or below this in an int variable will cause overflow or underflow

```
int i=32768; /*Overflow, the value is -32,768*/
int i=-32769; /*Underflow, the value is 32,767*/
```

Type Casting

- A data of one type can be assigned to a variable of another type. This is called typecasting

Example

```
int j=25;
flaot a;
a=j;
```

- Here j is typecast to a float
- The typecast operator can be used for better readability

Example

```
a=(float)j;
j=(int)a;
```

Console Input Operation

- A function scanf available in the header file stdio.h is used to perform console input

General Form

```
scanf("Format String",addresses of variables);
```

- String is a collection of characters. Format string will specify the type of the data to be read
- Following table shows the most commonly used format strings

Data Type	Format String
Char	%c
unsigned char	%c
Int	%d
short int	%d
unsigned short int	%u
long int	%ld
unsigned long int	%lu
Float	%f
Double	%lf
long double	%Lf

Example

```
scanf("%d",&j);
scanf("%d%d%d",&j,&k,&l);
scanf("%d%f",&a,&j);
```

- The data typed in through the keyboard can be separated by a space or an enter

Console Output Operation

- A function printf is available in the header file stdio.h is used for console output operation

General Form

```
printf("Format String",list of variable);
```

Example

```
printf("%d",j);
printf("%f",a);
```

- A string can be printed as follows

```
printf("Hello World");
```

- Constants and variables can be combined and printed as follows

```
printf("The sum of %d and %d is %d",i,j,k);
```

- Escape Sequences are used to print special characters

Escape Sequence	Special Character
\n	New Line
\b	Back Space
\f	Form Feed
\'	Single Quote
\"	Double Quote
\\	Back Slash
\t	Tab
\r	Carriage Return
\a	Alert

Example

```
printf("He said, \"Hello\"\n");
will print He said "Hello"
```

A Complete C Program

```
#include <stdio.h>
main(){
  printf("Hello World\n");
}
```

Arithmetic Operations

- Arithmetic Operators +, -, *, / and % are available for performing simple arithmetic operations. We can create arithmetic expressions using constant, variables and arithmetic operators

Example

```
a+b
x+35-y
```

- Be careful when you use / with integers, for it will yield only the integer part of the result. 5/2 will give 2 and not 2.5
- % works with integers only
- The ++ operator and -- operator are available to increment and decrement numeric values
- j++ will increment the value of j and j-- will decrement the value of j
- The increment operator and the decrement operator can be used in two forms - prefix and postfix
- The postfix form k=j++; will assign the value of j to k and then increment j
- The prefix form k=++j; will increment the value of j and then assign j to k
- Operator -- also works in a similar way
- Arithmetic Assignment Operators +=, -=, *=, /= and %= can be used for doing arithmetic operation and an assignment operation

Examples

- The expression a+=b is equivalent to a=a+b and a*=10 is equivalent to a=a*10

Logical Operations

- Relation operators <, >, <=, >=, == and != can be used to form relational expressions that will give either true or false as the answer

Example

```
a>b
x!=35
```

- Logical operators &&, || and ! are used to form logical expressions

Example

```
a>b && a>c
a>b||a>c
```

If Statement

- If statement is used in C to make a decision based on a logical expression

General Form 1

```
if (logical expression)
  statement;
```

General Form 2

```
if (logical expression)
  statement1;
else
  statement2;
```

Example

```
if (a>b)
  printf("%d\n",a);
if (a>b)
  printf("%d\n",a);
else
  printf("%d\n",b);
```

- Format the if statement properly to improve the readability
- There is no true and false in C. Integers are used to represent true and false. False is represented by 0 and true is represented by any non-zero integer. So the expressions a>b, x<=35 etc will produce integers only

The ?: operator

- The ?: ternary conditional operator has the following general form

General Form

```
Logical Expression?Expression1:Expression2
```

Example

```
j=a>b?1:2;
j=a>b?a>c?1:2:3;
```

- The ?: operator associates from right to left

Repetitive Statements

- C has repetitive statements that can be used in two situations – when you know and when you do not know the number of times of repetitions

The for Statement

- It is easy to use the for statement, when we know the number of times of repetition

General Form

```
for(expression1;logical expression;expression2)
  Statement;
```

- First, expression1 is evaluated. Then the logical expression is evaluated. If it is true, statement is executed. After that, expression2 is evaluated and the logical expression is checked again. This will continue till the logical expression becomes false
- This statement is bested suited when we know the number of times of repetition because, expression1 can be used to initialize a counter, expression2 can be used to increment or decrement the counter and the logical expression can be used to check the final value of the counter

Example

```
for(int i=1;i<=10;++i)
  printf("%d\n",i);
```

The while Statement

- The while statement can be used, when we do not know the number of times of repetition

General Form

```
while(logical expression)
  Statement;
```

- The statement will be repeated as long as the condition is true

Example

```
while(a<10)
  printf("%d\n",a++);
```

The do-while Statement

- The do-while statement is very similar to the while statement. In the while statement, the logical expression is evaluated before executing the statement, but in do-while, the statement is executed before evaluating the logical expression

General Form

```
do
  Statement;
while(logical expression);
```

Example

```
do
  printf("%d\n",a++);
while(a!=0);
```

The break Statement

- The break statement is used to exit from a repetitive block even if the logical expression is true

General Form

```
break;
```

Example

```
for(int k=1;k<10;++k){
  if (k==5)
    break;
  printf("%d\n",k);
}
```

The continue Statement

- The continue statement will ignore the current iteration and continue with the next iteration in a repetitive block

General Form

```
continue;
```

Example

```
for(int k=1;k<10;++k){
  if (k==5)
    continue;
  printf("%d\n",k);
}
```

The switch Statement

The switch statement can be used to branch to different places based on the value of an expression

General Form

```
switch(expression){
  case value1:
    Set of Statements 1
  case value 2:
    Set of Statements 2
  case value 3:
    Set of Statements 3
  case value n:
    Set of Statements n
  default:
    Set of Statements x
}
```

- If the expression evaluates to value 1, the program jumps to Set of Statements 1 and continues execution from there. If it is value 2, the program jumps to Set of Statements 2 and so on
- The default section is optional. If the expression is not evaluating to value 1, 2, 3 or n, then Set of Statements x will be executed

Example

```
switch(a){
  case 10:
    printf("Hello\n");
  case 20:
    printf("Welcome\n");
  case 7:
    printf("Ok\n");
  default:
    printf("Bye\n");
}
```

- The break statement can be used to come out of a switch block also

Functions

- Functions are used for modular programming
- Functions make the program easy to write, read, understand, debug and maintain
- Functions make the program smaller

- Functions help to create reusable modules
- In C and C++ we have functions that can return a value and functions that do not return any value

General Form

```
Return_type function_name(List of Parameters){
}
```

Examples

```
void sample(){
  Statements
}
float test(float x, int n){
  Statements
}
```

- Each function in a program should have a unique name in C
- A void function is called by giving the name of the function, as if it is a statement
- If a function is taking some parameters, they should be passed to the function
- The data that we pass to the function are called arguments and the variables in which they are received by the function are called formal parameters
- The arguments and the formal parameters should match in data type, number and order
- A variable that is declared inside a function can be used only by that function and is known as a local variable of the function
- A variable that is declared outside all the functions can be used by all the functions in the program and is known as a global variable

The return Statement

- The return statement can be used to return a value from a function

General Form

```
return expression;
```

Example

```
return;
return a;
return 100;
return a+b-c;
```

- When we are calling a function that returns a value, the returned value can be collected in a variable
- The returned value can be used for other purposes also like printing and creating expressions

Example

```
a=square(b);
printf("%d\n",square(q));
k=square(x)-square(y);
m=square(square(n));
```

- Generally we can call a function that is returning a value by giving the name of the function in a position, where we can have a return type constant
- That is, if a function is returning an integer, the name of that function can be given in a position where an integer constant can be given
- A function that returns a value can be even called as if it is a void function, ignoring the returned value

Pass by Value

- When an argument is passed to a formal parameter, the value of the argument is copied to the formal parameter
- Since the formal parameter is only a copy of the argument, any change made on the formal parameter will not affect the argument
- This is called pass by value

Automatic and Static Variables

- A local variable declared in a function is automatically created when the function is called. It is automatically destroyed when the function ends. So they are called automatic variables
- A local variable, declared as static, will be created when the function is called for the first time. It will not get destroyed when the function ends. It will get destroyed only when the whole program ends
- A static data is automatically initialized to zero
- Automatic and static are two storage classes

Function Declaration and Definition

- There can be any number of functions in a program and they can be written in any order
- The program will always start with main
- A function should be at least declared before calling it. It can be defined later

Recursive Functions

- A function can call itself. Such functions are called recursive functions

Storing the Functions

- The functions in a C program can be stored in different files
- A global variable of a file can be accessed as an external variable in another file
- But a static global variable cannot be accessed as an external variable
- Extern is a storage class

Data Structures

Data structures make storage of data easier
A normal variable is the simplest form of a data structures

Arrays

- Array is a data structure we use when we want to store a large quantity of data of the same type
- Arrays are declared as int a[10]; float x[20]; etc
- In an array int a[10], there are 10 elements - a[0] to a[9]

- The elements of an array will occupy consecutive memory locations
- When you are declaring an array in C, the size of the array should be a specified as a constant
- The is no boundary checking for an array in C
- An array can be initialized as follows

```
int a[5]={5,67,23,5,8};
int a[]={5,67,23,5,8};
int a[5]={5,6};
```

- An array cannot be assigned to another array

Character Arrays or Strings

- Character arrays are terminated by a null character
- A character array can be read using the function scanf using the format string %s
- Similarly the function printf can be used for printing a character array
- Functions like gets and puts are also available for reading and writing character arrays
- Character arrays can be initialized as follows

```
char a[6]={'H', 'E', 'L', 'L', 'O', '\0'};
char a[6]={"HELLO"};
char a[]={"HELLO"};
```

Array of Arrays

- When we want to have several arrays we go for an array of arrays
- An array of array is declared as int a[3][4];
- Here a is an array containing 3 elements – a[0], a[1] and a[2]. Each of this is an integer array containing 4 integers

Arrays and Functions

- Arrays can be passed to a function as arguments. We have to specify only the name of the array as the argument
- The formal parameters that are declared to receive arrays are declared just like the argument arrays. But their size is optional. This is applicable to array of arrays also
- When the formal parameter is changed, the actual argument will change in the case of an array. This looks like pass by reference
- Normally we cannot return an array from a function

Structures

- Structures are used when we want to have a compound data type that is made up of several components

Example

```
struct student{
  char name[20];
  int classno;
  int marks;
};
```

- Now, a structure called struct student is created. This can be treated as a data type and we can create variables of this type

Example

```
struct student a, b;
```

- The individual elements can be accessed using the **.** operator as a.name, a.classno etc
- The elements of a structure will occupy consecutive memory locations
- A structure variable can be initialized as follows

```
struct student a={"Tom",12,45};
```

- We can have arrays in a structure and array of structures
- We can also have a structure variable of one type as a member of another structure
- A structure variable can be assigned to another structure variable using the assignment operator
- A structure variable can be passed to a function
- A structure variable can also be returned from a function
- Structure variables can be declared along with the structure declaration as follows

```
struct student{
  char name[20];
  int classno;
}a, b;
```

- In such cases, we can even create nameless structures as follows

```
struct {
  char name[20];
  int classno;
}a, b;
```

Union

- Unions are very similar to a structure in syntax, but the fields of a union will share the same memory location

Example

```
union test{
  int k;
  double d;
  char c;
};
union test u;
```

- Here u.k, u.d and u.c will occupy the same memory locations
- A union can be used to store mutually exclusive data

Pointers

- A pointer is a variable that can hold the address of another variable
- A pointer can be declared by adding * with the variable
- So, int *p; means, p is an integer pointer and char *q; means, q is a character pointer
- The address of a variable can be stored in a pointer as follows

```
int a, *p;
p=&a;
```

- Now, we can use p instead of &a and *p instead of a

Pointers and Functions

- Pointers can be passed to a function to bring about the effect of pass by reference
- A function can return a pointer, but never return a pointer to a local variable
- We can also have a pointer to a function

Pointers and Arrays

- The name of an array is a pointer to the array
- When we say a[3], it means *(a+3)
- Pointer arithmetic is different from normal arithmetic
- An element in an array of array is also located using the same technique
- Since the name of an array is a pointer to the array, when we try to pass an array to a function, we are actually passing a pointer to the array
- That is why we get the effect of pass by reference when arrays are passed to the function. Same is reason for not giving the size of the array in the formal parameter
- We can have an array of pointers

Pointers and Structures

- We can have a pointer to a structure as follows

```
struct student a, *p;
p=&a;
```

- Now, instead of a.classno, we can say (*p).classno or still better, p->classno
- Pointer to a structure can be used to pass a structure variable to a function to get the effect of pass by reference
- Pointer to a structure can be used for creating data structures like linked lists

Pointers and Unions

- Pointer to a Union is similar to a pointer to a structure

Pointer to a Pointer

- We can have a pointer to a pointer as follows

```
int a, *p, **q;
p=&a;
q=&p;
```

Pointers and System Level Programming

- Pointers are used for system level programming as they can be used to directly communicate to the hardware

Pointers and Dynamic Memory Allocation

- Dynamic variables are not declared as normal variables, but are created as the program is running
- They cannot be given any name, but can be accessed using pointers
- They can be and should be deallocated after use

Preprocessor Directives

- A program can be manipulated using the pre processor, before it reaches the compiler
- Pre processor directives are used for this purpose

- #define - Macros (#defines) can be used to declare constants, improve the readability of a program, to write programs that are faster than functions and to write generic programs
- #ifdef, #ifndef, #if, #else, #endif

The typedef Statement

- The tyedef statement can be used to improve the readability of a program

Example

```
typedef int COUNT;
typedef char STRING[20];
typedef struct{
  char name[20];
  int classno;
  int marks;
}STUDENT;
```

- The typedef statement is very frequently used to improve the readability when complicated pointer types are involved

Example

```
typedef void (*X)(int);
X p[10];
```

Files

- Files are used to store data permanently
- There are three major file operations – Opening the file, Reading data from or writing data to the file and closing the file. Library functions are available in C for doing these
- The functions fopen, fscanf, fprintf, fclose, feof etc that are available in stdio.h are used for file operations

The enumerated Data Type

- It helps the user to create his own data types

Example

```
eum color {black, white, green, red, blue};
```

- A new data type enum color is created. We can declare variables of this type
- The constants available in this type are black, white, green, red and blue
- The constants have the following relationship between them
  black<white<green<red<blue
- Enumerated data types are used to improve the readability of a program

EXERCISE

1. Find the largest of three numbers
2. Print all multiples of 7 between 1 and 100
3. Solve $y = x - x^3 + x^5 - x^7 + \ldots\ldots\ldots x^n$
4. Find the largest and second largest number from a set of n numbers
5. Print all prime numbers between 1 and 100.
6. Find the LCM of two numbers
7. Find the HCF of two numbers
8. Find whether a number is palindrome or not
9. Find whether an year is a leap year.
10. Find the roots of a quadratic equation.

OOP through C++

Some new features in C++ language when compared to C

Comments

- Multiline comment

```
/* Comment statements
   …..
*/
```

- Single line comment

```
// This is a single line comment
```

Console Output

Standard output stream object

Insertion operator / Put-to operator

```
cout << "Welcome…";
```

Examples:

```
cout << "Hello," << " Goodmorning";
cout <<  "Sum = " << iResult;
cout << "Hai\nThere";
cout << iNum1 << " + " << iNum2 << " = " << iSum;
```

Console Input

Standard input stream object

Extraction operator / Get-from operator

```
cin >> var;
```

Examples:

```
cout << "Enter Number: ";
cin >> iNum;
cin >> iNum1 >> iNum2;
```

iostream.h

- It contains the declarations for cout, <<, cin and >>
- So include it in your program

```
#include <iostream.h>
```

Declaration of variables

- C++ allows the declaration of a variable anywhere in the scope.
- But before or at the place of its first use.

```
int iNum1, iNum2;
clrscr();
cout << "Enter two numbers";
cin >> iNum1 >> iNum2;
int iSum;
iSum = iNum1 + iNum2;
for(int i = 0; i < 5; i++)
{   }
```

for loop block scope

```
int x = 50;
for(int i = 0; i < 3; i++)
{
        int x = 10;
}
cout << i << "\n" << x;
```

Output:

```
3
50
```

Reference variables

```
datatype &reference-name = variable-name
int x = 100;
int &y = x;
```

- A reference variable provides an alias (another name) for a previously defined variable.
- More efficient memory usage.
- Can replace pointers to some extend.
- Call by reference.

Dynamic Memory management operators

new

Used to dynamically allocate memory

```
pointer-variable = new data-type;
```

- Here, *pointer-variable* is a pointer of type *data-type*

- *new* operator allocates sufficient memory to hold a data object of type *data-type* and returns the address of the allocated memory.

Egs:
int *p = new int; // Allocates memory for 1 integer
int *q = new int(50); // Allocates memory for 1 integer and initialize it to 50
int *r = new int[10]; // Allocates memory for 10 integer contiguously

delete
Used to release memory which was allocated dynamically
delete pointer-variable;

- Here, *pointer-variable* is a pointer which was previously allocated memory using new operator.

Egs:
delete p; // Releases memory used by p
delete [] r; // Releases memory created for the array

Object oriented programming

Procedure Oriented Approach

- Focus is on procedures
- No protection for data
- Data may be shared for convenience
- More difficult to modify
- Hard to manage complexity
- Hard to maintain when program size increased
- Top-down approach

The object oriented approach

- It allows you to decompose a problem into a number of entities called objects and then build data and functions around these entities.
- Here more importance is given to the data rather than the functions.
- Bottom-up approach

What are the steps?

1. Identity the entities in your domain
2. Identify the attributes of each entities
3. Identify the methods that the entities can perform
4. Identify the relationship between the entities

Basic OOP Concepts

- Abstraction
- Encapsulation
- Polymorphism
- Inheritance

Abstraction

It is what you conceive about an entity.
Eg: What can you say about a computer mouse?

Have two buttons
Have a scroll button
It is optical
Black in color

- This is your abstraction about the mouse.
- Another person may describe this same mouse in a different way, which is his abstraction of the mouse.

Encapsulation

- Binding of data and the methods that operate on the data together as a single unit (capsule) is known as encapsulation.
- This unit is referred to as *class*.

Class Vs Object

- Class:
  - A category or group of things that have similar attributes and common behaviors.
- Object:
  - A specific thing which has specific values for the attributes and behavior.

Using Encapsulation

- Data and methods are bound together tightly
- Data Hiding
  - It is like a black box
  - Data and methods of the class is hidden from outside
  - Nothing outside the class can access the data or methods unless permitted by the class
- Communication between the objects is through messaging (methods).

Polymorphism

- Ability to take more than one form
- An operation may exhibit different behavior in different situations
- The behavior depends on the types of data used in the operation

Inheritance

- The children will be inheriting the behaviors of the parents
- They will have there own behaviors as well

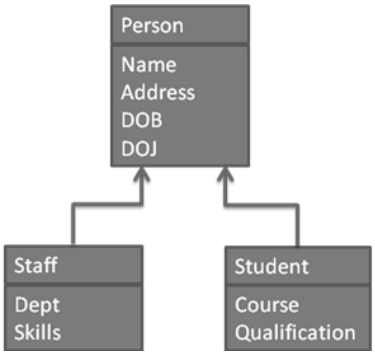
A Problem: to automate TurboPlus Information Technologies

- You will be able to readily identify entities
  - Staff
  - Student

Staff
Name Address DOB DOJ Dept Skills

Student
Name Address DOB DOJ Course Qualification

- Some common features are present in both the entities. You can refine the entity by taking the common features outside



Advantages of using OOP

- Modularity
  - Large software projects can be split up in smaller pieces.
- Reusability
  - Programs can be assembled from pre-written software components.
- Extensibility
  - New software components can be written or developed from existing ones.

C++ implementation of oop

Class

- Can be considered as an extension to structures.
- Supports the implementation of all the OOP features.
- An Abstract Data Type (ADT)

Specifying a Class

Consists of

- Class declaration
  - Describes the type and scope of its members
- Class method definitions
  - Describe how the class methods are implemented

class *class-name*
{
    variable declarations;
    method declarations;
};

An Example

```
class Student
{
        int iRegno;
```

Data members

```
        char sName[30];
        char sCourse[20];

        void AddStudent();
        void EditStudent();
        void DeleteStudent();
        void SearchStudent();
        void DisplayStudent();
};
```

Member methods

Defining Member Methods

- In C++ you can define the member methods
  - outside the class definition OR
  - inside the class definition
- When you define it inside the class definition, it is by default considered as *inline methods*.
- When you define it outside the class definition, you have to specifically make it inline (if desired).

Outside the class definition

return-type class-name :: *function-name*(*argument declarations*)

```
{
        // function body
}
```

Eg:

```
void Student :: AddStudent()
{
        …
}
```

Inside the class definition

```
class Student
{
        int iRegno;
        char sName[30];
        char sCourse[20];

        void AddStudent()
        {
                …
        }
        void EditStudent()
        {
                …
        }
};
```

Creating Objects

class-name object-name;

Eg:

```
Student stud1;
```

Accessing Class Members

- Use dot (.) operator along with the object-name.

```
stud1.iRegno;
stud1.sName;
```

Data hiding

- By default all the members of class in C++ are *private* to the class.
- You can give access to members from outside the class by using the access specifier, *public*.

```
class Student
{
        int iRegno;
        char sName[30];
    public:
        void AddStudent();
        void DispStudent();
};
```

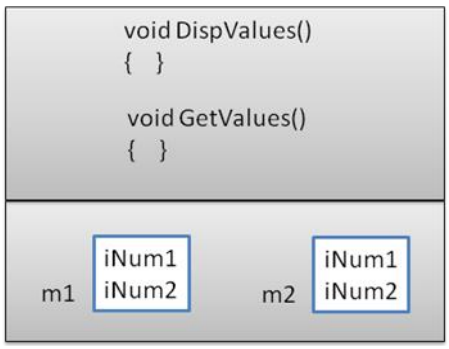
Access specifiers in C++

- private
- public
- protected

Memory allocation for objects

- For each object separate memory will be allocated for data members.
- All the methods will be allocated memory only once and is shared by all the objects created from that class.
- Eg:

```
class Mem
{
        int iNum1, iNum2;
        void GetValues()
        {  …  }
        void DispValues()
        {  …  }
};
Mem m1, m2;
```



this pointer

- When an object is used to call a method, implicitly a pointer called *this pointer* is passed into the function, which will contain the address of the object which invoked the function.
- So inside the method,

```
void DispData()
{
        cout << x;   ⟺   cout << this -> x;
}
```

Array of Objects

- To store more than one object of the same class and to manipulate them easily, an array of objects can be created.
- *class-name obj*[*size*];
- Student stud[10];
- stud[2].iRegno = 20;

Method Overloading

- In C++, you can have more than one method with same method name but with different number / type of parameters.
- This is one method by which polymorphism is implemented in C++.

```
class DataStore
{
    int iData;
  public:
    void SetData()
    {
        cout << "Enter data: ";
        cin >> iData;
    }
    void SetData(int data)
    {
        iData = data;
    }
};
void main()
{
    DataStore ds1, ds2;
    ds1.SetData();
```

```
    ds2.SetData(100);
}
```

Objects as method arguments

- You can pass arguments of any type including the same class type to member methods.
- In C++ objects can be passed by value or reference.
- When you invoke a member method using an object, that object in implicitly passed into the function *by reference*.
- When you pass objects as arguments to member methods, it is by default *passed by value*.
- You can return any type including object of the same type from member methods by value or by reference.

Constructor

- A special member method invoked automatically when an object of the class is created
- Used to construct an object in memory
- The compiler provides a built-in default constructor if you don't have any constructors defined in the class.

Constructor – special features

- Cannot have any return type (not even void)
- Name of the method is same as that of the class
- Can have arguments
- Will be automatically invoked when an object is created

Default constructor

- A constructor with no parameters in known as a default constructor

```
class Integer
{
        int iVal;
    public:
        Integer()
        {
                iVal = 0;
        }
};
Integer int1;
```

- This constructor can be used to give initial values for class data members.

Parameterized Constructors

- Constructor that can take parameters

```
class Integer
{
        int iVal;
```

```
    public:
            Integer(int x)
            {
                    iVal = x;
            }
};
Integer int1(10);
Integer int2 = Integer(20);
Integer int3 = 30;
```

- This constructor can be used to initialize different objects with different values.

Copy Constructor

- A special type of parameterized constructor which can be used to initialize an object with the data of another object.

```
class Integer
{
            int iVal;
    public:
            Integer(int x)
            {    iVal = x;    }

            Integer(Integer &x)
            {
                    iVal = x.iVal;
            }
};
Integer i1(100);
Integer i2(i1);   or   Integer i2 = Integer(i1)   or   Integer i2 = i1;
```

Some Points

- Constructors can be overloaded.
- When defining copy constructors, the object is to be passed by reference.
- If you have atleast one constructor defined inside the class, the built-in default constructor *will not* be invoked.
- It is a good programming practice to give your own default constructor in the class even if it is not needed.

Destructor

- A special member function invoked automatically when an object is being destroyed.
- If no destructor is defined in the class the compiler supplies a built-in destructor.

Destructor – special features

- Has the same name as that of the class with a tilde(~) at the beginning.
- Cannot have a return type.
- Cannot take any arguments.

- If you are creating objects dynamically (using new operator), it is good to define destructor which will release the dynamically created memory (using delete operator).

Static Data Member

- Data members of a class can be declared as *static*.
- Only one copy of the static member will be created for the entire class and is shared by all the objects of that class.
- The type and scope of each static data member must be defined outside the class because static data members are stored separately rather than as part of any object.
- If no initial value is specified, it is initialized to zero.
- Also known as *class variables*.
- Can be accessed using class name.
- Static members are usually used to maintain values common to the entire class.

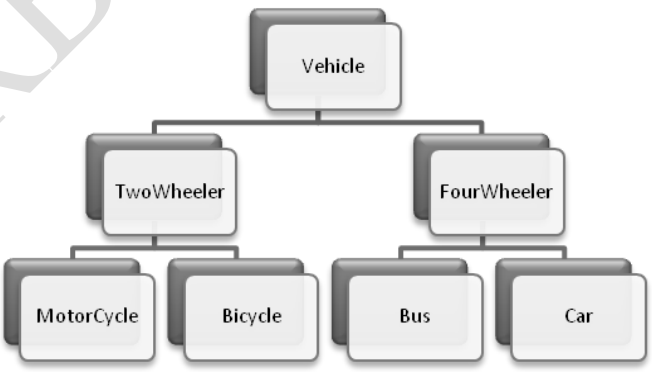
```
class Item
{
   public:
      static int iCount;
        int iNum;
};
int Item :: iCount = 1;
main()
{
       cout << Item :: iCount;   // Accessing using class name
}
```

Static member methods

- A static member method can access only static members of the same class.
- A static member method can be accessed using the class name.

Inheritance

- The mechanism of deriving a new class from an older class.
- The class from which derivation takes place is called the *base class*.
- The class into which derivation takes place is called the *derived class*.
- The derived class derives some or all traits (features) of the base class.

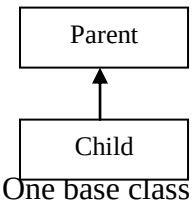


Implements ***is a*** relationship

Types of inheritance in C++

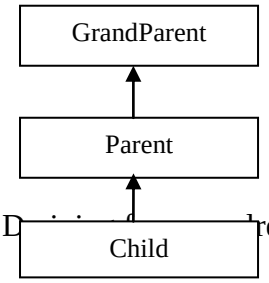
- Single inheritance
- Multi-level inheritance
- Hierarchical inheritance
- Multiple inheritance
- Hybrid inheritance

Single Inheritance



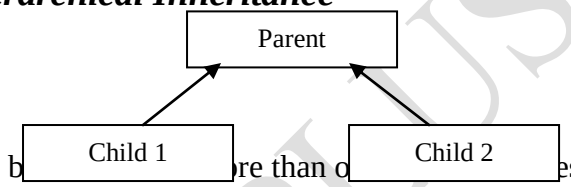
One base class and one derived class.

Multi-level Inheritance



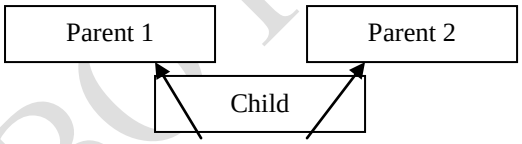
D ready derived class.

Hierarchical Inheritance



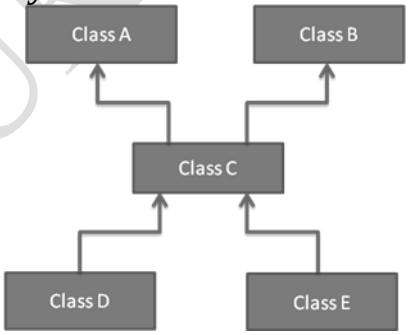
One b re than o es.

Multiple Inheritance



More than one base class and one derived classes.

Hybrid Inheritance



Some combination of the basic inheritance types.

Defining Derived Class

class *derived-class* : *visibility-mode base-class*
{
……. // derived class members
}

- The default *visibility-mode* is *private*.

Example

```
class Person
{
        char sName[20];
        char sAddress[50];
        char sDOB[15];
    public:
        void GetPerson();
        void ShowPerson()    …
};

class Student : public Person
{
        int iRegno;
        char sCourse[20];
    public:
        void GetStudent();
        void ShowStudent();  …
};
```

Single Inheritance

```
class A
{
        …
};
class B : public A
{
        …
};
```

Protected

- Private member of a class cannot be inherited.
- Therefore it is not available for the derived class directly.
- A member declared as *protected* is accessible within that class and any class immediately derived from it.
- It cannot be accessed by the functions outside these two classes.

Visibility of Inherited members

Base class	Derived class visibility

visibility	public derivation	protected derivation	private derivation
public	public	protected	private
protected	protected	protected	private
private	Not inherited	Not inherited	Not inherited

Multilevel Inheritance

```
class A
{
                …
};
class B : public A
{
                …
};
class C : public B
{
                …
};
```

Hierarchical Inheritance

```
class A
{
                …
};
class B : public A
{
                …
};
class C : public A
{
                …
};
```

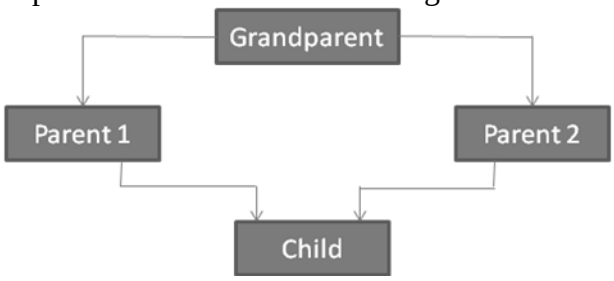
Multiple Inheritance

```
class A
{
                …
};
class B
{
                …
};

class C : public A, public B
{
                …
};
```

Problems with Multiple inheritance

Due to getting the features from different classes, there may be situations where multiple copies of the same feature coming to the derived class.



So when an object of the *Child* class is used to invoke a *Grandparent* class method (which is derived more than once through different paths), the compiler will not be able to decide which copy to call. This is an ambiguity situation.

Solution: Virtual Base Class

- The duplication of inherited members due to multiple paths can be avoided by making the common class (Grandparent) as *virtual base* class while declaring the intermediate base classes.

```
class Grandparent
{  …  };
class Parent1 : public virtual Grandparent
{  …  };
class Parent2 : virtual public Grandparent
{  …  };
class Child : public Parent1, public Parent2
{  …  };
```

Constructors in derived classes

- In single, multilevel and hierarchical inheritance, always the base class constructors are executed first from top to bottom.
- In multiple inheritance, the base class constructors are called in the order in which they appear in the declaration of the derived class.
- When a base class need some data in the constructor it has to be explicitly passed from the derived class constructor.

Abstract class

- Is a class which contains at least one *pure virtual function*.
- You cannot create an object of an abstract class.
- Used for just inheritance.
- Can have other normal members.

```
class Shape
{
    protected:
            int Dim1, Dim2, Dim3;
```

```
        float Area;
    public:
        void GetDims(int D1 = 0, int D2 = 0, int D3 = 0)
        {   Dim1 = D1;   Dim2 = D2;   Dim3 = D3;   }
        virtual void FindArea() = 0;   // pure virtual function
        void ShowArea()
        {   cout << Area;   }
};
```

Pure Virtual function

- A function for which no definition but only declaration is given in the class.
- The function is declared as *virtual* and is assigned to 0.
  Eg: virtual void FindArea() = 0;
- Such a declaration just functions as a place holder for the actual function.
- Any class which derives from a class which has pure virtual function (abstract class) must give definition to the function.
- Each derived class can give its own implementation for the function.
- If the derived class does not implement the pure virtual function, then that class also will become an abstract class.

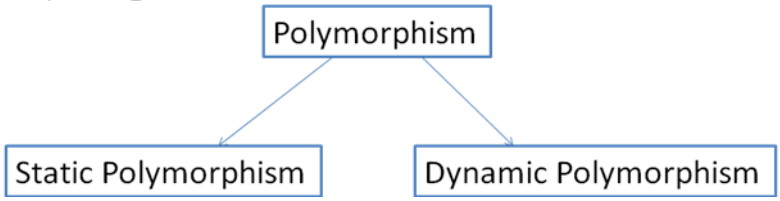
Different Implementations

```
class Circle : public Shape
{
        void FindArea()
        {   Area = 3.14 * Dim1 * Dim1   }
}
class Rectangle : public Shape
{
        void FindArea()
        {   Area = Dim1 * Dim2   }
}
```

Nested Classes

- A class containing the objects of other classes as its members is called a *nested class*.
- It is also called *containership*.
- This implements ***has a*** relationship.
- When an object of such a class is created, the nested objects are created first, then the container object part.
- So the container class's constructors should explicitly pass the necessary arguments to the contained object's constructors.

Polymorphism



Method Overloading	Method Overriding
Operator Overloading	+
	virtual methods

Object Pointers

- Can be used to create objects at run time.

```
class A
{
     int a;
    public:
     void ShowA()
     {
              cout << a;
     }
};
void main()
{
     A *ptrA;
     ptrA = new A;
     ptrA -> ShowA();
}
```

Using Base class Pointer

- If you have a base class pointer, that pointer can point to
  - any of the objects of its own class
  - any of the objects of classes derived from it

```
class Base
{   };
class Derived1 : public Base
{   };
class Derived2 : public Base
{   };
class Derived3 : public Derived1
{   };
Base *ptr;
Base b1;
Derived1 d1;
Derived2 d2;
Derived3 d3;
ptr = &b1;
ptr = &d1
ptr = &d2;
ptr = &d3;
```

- When you use a base class pointer, which is pointing to a derived class object, to access any of the derived class members, it can access only those members which are derived from the base class whose pointer you are using.

Example

```
class Base
{
        int x;
    public:
        void ShowBase()
       {   }
};
class Derived : public Base
{
        int y;
    public:
        void ShowDerived()
        {   }
};
void main()
{
     Base *ptr;
     Derived d1;
     ptr = &d1;
     ptr -> ShowBase();    // OK
     ptr -> ShowDerived();  // Error
}
```

Another Example

```
class Base
{
        int x;
    public:
        void Show()
        {   cout << x;    }
};
class Derived : public Base
{
        int y;
    public:
        void Show() // method overriding
        {   cout << y;   }
};
void main()
{
     Base *ptr;
     Base b1;
     b1.Show();     // shows x
     ptr = &b1;
     ptr -> Show();    // shows x
     Derived d1;
     d1.Show();     // shows y
     ptr = &d1;
     ptr -> Show();    // shows x
}
```

Dynamic Polymorphism

- Also called *Runtime Polymorphism* or *Late Binding.*
- Here the compiler will not be able to link the function that is to be called during compile time.
- At runtime, when it is known what class objects are in consideration, the appropriate version of the function is called.

Implementation

```
class Base
{
        int x;
    public:
       virtual void Show()
        {   cout << x;    }
};
class Derived : public Base
{
        int y;
    public:
       void Show() // method overriding
       {   cout << y;   }
};
void main()
{
     Base *ptr;
     Base b1;
     b1.Show();    // shows x
     ptr = &b1;
     ptr -> Show();   // shows x
     Derived d1;
     d1.Show();    // shows y
     ptr = &d1;
     ptr -> Show();   // shows y
}
```

Virtual Functions

- If a function in a base class definition is declared to be virtual, and is declared exactly the same way in one or more derived classes, then all calls to that function using pointers or references of type *base class* will invoke the function that is specified by the object being pointed at, and *not* by the type of the pointer itself.

EXCERCISE

1. Create a class Date for storing a date. Write the following functions. (i) Function to read the date from the keyboard (ii) Function to check the validity of the date (iii) Function to print the date on the screen (iv) Function to add a specified number of days to the date
2. Create a class String to hold a character string. Write the following functions. (i) Function to initialize string (ii) Function to copy string (iii) Function to compare strings considering the case (iv) Function to concatenate strings. The String object that calls the function should not change (v) Function to concatenate strings. The String object that calls the function should change (vi) Function to print the string on the screen

3. Write a program that reads a C++ program from the keyboard. Remove all the // and /**/ comments from the program and print the program on the screen. Beware of the // and /**/ that come inside a string
4. Create a class Brush. Write the following functions. (i) Function to draw a circle (ii) Function to draw a rectangle (iii) Function to choose the color of drawing
5. Create a class Calculator. Write the following functions. (i) Function to add two numbers (ii) Function to find the square root of a number (iii) Function to find the sine of an angle
6. Create a class Integer to store an int. Write the following functions. (i) Function to initialize an Integer object with an int (ii) Function to convert an Integer object to a binary string (iii) Function to convert an Integer object to a hexadecimal string (iv) Function to add two Integer objects
7. Overload the prefix and postfix ++ and – operators in the class Date
8. Overload the operators +, +=, <, >, <=, >=, == and ! = in the class String
9. Create a class Complex to store a complex number. Overload the operators +, -, *, /, +=, -=, *= and /=
10. Write a class IntegerString to store an integer as a string. Eg : "156". Overload the operators +, -, *, /, %, +=, -=, *=, /=, %=, ++ (prefix and postfix) and – (prefix and postfix)
11. Create a class Window. Write functions to set the width and height of the window and display the window. Create a class that has all the properties of class Window. But while displaying the window, the new class should show a shadow also
12. Create a class List to store a list of integers. Hint – Use an integer array inside the class. Write the following functions (i) Function to add an element at a particular position in the list. Note that the list should not have blank elements in between elements. That is the elements should be in contiguous cells (ii) Function to delete an element at a particular position in the list (iii) Function to display the list (iv) Function to get the number of elements in the list (v) Function to check whether the list is empty (vi) Function to check whether the list is full
13. Create a Stack to store integers. Use the class List for implementing the stack
14. Create a class Book to store the Title, Name of Author, Price and Number of Pages. Also create a class CD to store the Title, Name of Author, Price and Total Play Time. Include functions in both the classes to input and output the details
15. Write a program to store some circles and rectangles
16. Write a program to create a linked list. Include functions in question 12
17. Repeat program 13. Use a linked list instead of the list created by program 12 and show that the class Stack will not change
18. Modify the class Stack that uses the Linked List so that it throws exception whenever there is an overflow or underflow
19. Modify the class Stack of program 18, so that the stack can hold any kind of data
20. Write a program to store the Name, Class Number and Marks of n students. Write functions to display the details of all students and search for a student based on Class Number

21 Consider a school where there are a number of students. You have to develop an application that will read in the details of each student (regno, name, class), the marks got by them for three terms for 6 subjects and calculate and display the result. The result should display the regno, name, total marks and the grade of the each student when the class is given.
    a) Identify the classes for this application
    b) Identify the relationships between the classes.

c) Use inheritance where necessary.

Use a menu driven program to implement the above scenario.

22. Create a class TextBox using graphics functions which will implement the functionalities like:
    (i) Provision for specifying the position to display the textbox
    (ii) Provision for specifying the width of the textbox
    (iii) Should be able to give the maximum length of the text
    (iv) Read in text into the textbox
    (v) Should be able to specify the characters which can be used in the textbox
    (vi) Should be able to specify if password field or normal field; if password, should display only ***** in the textbox
    (vii) Provision for returning the text in the textbox
    (viii) Provision for clearing the textbox

    Use a program to test the class.
23. Create a class Crypto that can be used to encrypt and decrypt a text given to it. Use another class, Text which will inherit this class to do encryption and decryption.

24. Create a class Date that can be used to validate a given date to check if it is a valid date. It should have provisions to do all date manipulations like:
    a. to add number of days, months, years to a given date
    b. to subtract number of days, months, years from a given date
    c. find the difference between two dates in terms of days, months or years
    d. to compare two dates

    Use operator overloading and type-conversion functions for the above manipulations.