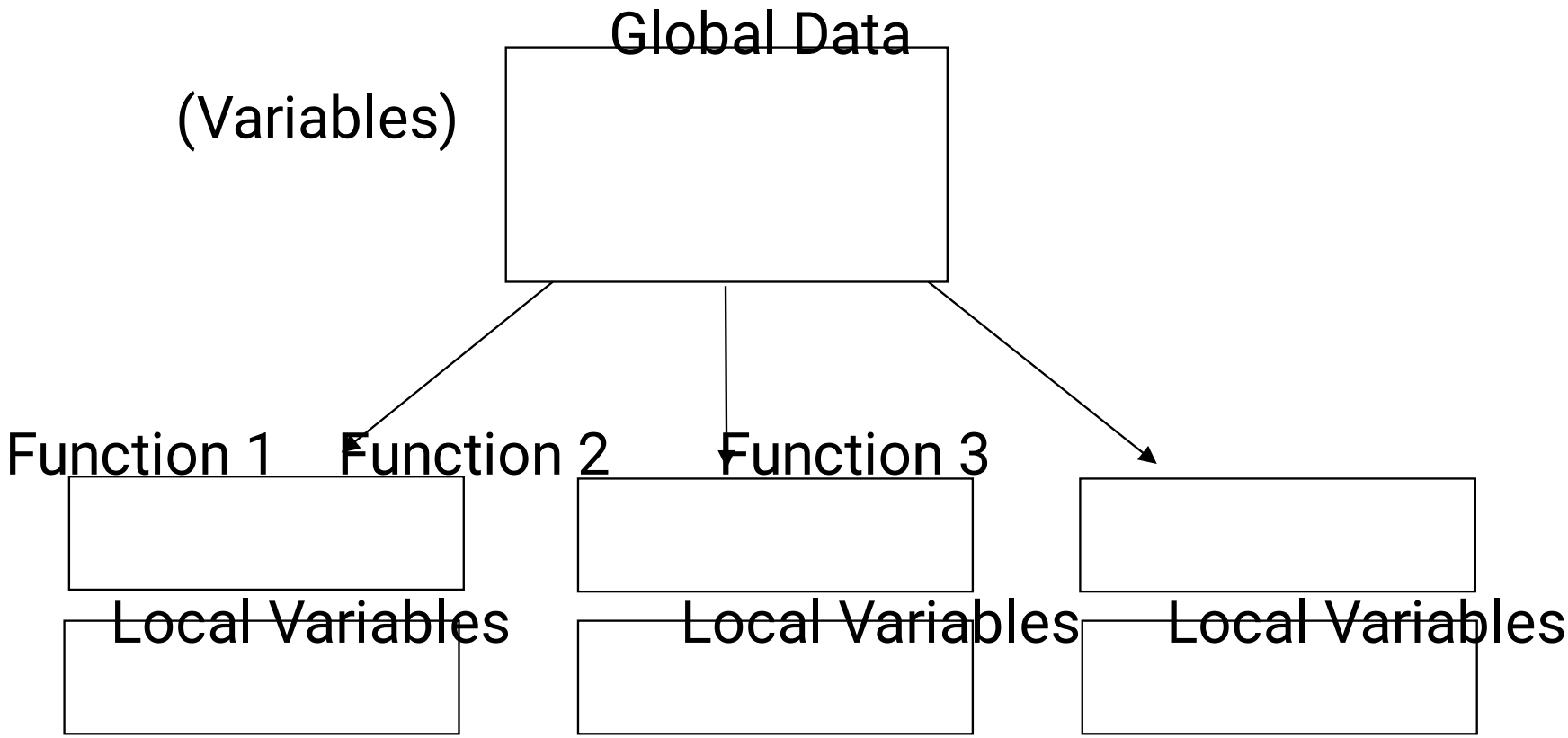
C++ and Object Oriented Programming
CS-14

Chapter 1: Principles of Object Oriented Programming

Procedure Oriented Programming :

- Meaning
- Examples : C,COBOL, FORTRAN,
- Idea of Local & Global variables
- Characteristics of POP Language :
  - Main focus on functions
  - Complex programs are divided into functions
  - Global data shared by functions
  - No protection of data, data can move around functions
  - Follows top-to-bottom approach


Global Data
(Variables)
Function 1
Function 2
Function 3
Local Variables
Local Variables
Local Variables

Object Oriented Programming (OOP) :

- Meaning
- Major modifications in POP
- Data is protected, can’t move around functions
- Complex program is divided into entities called **Objects**
- Associated with a Class and function outside class is not accessible to those data

Characteristics of OOP :

- Focus is on data rather than functions
- Large programs divided into objects
- Data is hidden, can’t be accessed outside functions
- Object passes information through functions
- Follows bottom-to-top approach

Difference between POP and OOP :

POP Language

- Large code divided into functions
- Focus on functions
- Function uses global data
- Top-to-bottom approach
- Ex: C,FORTRAN, COBOL

OOP Language

- Large code divided into Objects rather than functions
- Focus on objects
- Data is hidden. Can’t be accessed by outside functions
- Bottom-to-top approach
- Ex: C++, C#, Java

Basic Concepts of OOP :

- Objects
- Class
- Data Abstraction
- Encapsulation
- Inheritance
- Polymorphism

```
. class student
. {
        int rollno;
        char name[50];
        float per;
        void display();
. };
. Student s;
. S.rollno
. S.name
. S.per
. S.display();
```

Objects :

- Basic unit of OOP
- Also known as real-time entity
- It is a unit that bundles data and functions
- Represents anything such as a pen, a person, a book, etc…

```
 struct student
{
   int I;                                  s.age=50; s.i=10;
   float age;
}s;--- object
```

Classes :

- It is a template for its objects
- Represents category of its objects
- Ex: Apple is a member of class fruit, rose is a member of class flower
- Contains member (data) and Functions
- It is variable of its type (class)
- variable declaration : int i, same way :-> fruit apple
- Can be compared with Structures and Unions of C language with some updation

Data Abstraction :

- Also known as **data hiding**
- Only shows essential information
- Hides background details
- Example, how TV works

- int I;
- Integer I;

Encapsulation :

- Can compare with medicine capsule
- Main element is kept inside, covered by outside wrapper for protection
- Same way, data and variables are wrapped into a single unit known as **Class**
- Process of wrapping data and function into a single unit is called **Encapsulation**
- Data is not accessible from outside entities

```
. Class student
. {
      int a,b,c;
       Float d,e;
      void print();
. }
. Student s1;
. A; s1.a;
```

Polymorphism :

- Greek word Poly + morphis
- Poly = More than one
- Morphis = Form (swaroop)
- **Two** concepts inside this :
- Function (Method) overloading / Constructor overloading
- Operator overloading
- A single person can behave differently in different situations
- 2 + 3 = 5 but also, “abc” + “xyz” = abcxyz

. Void sum()
. {
. }

. Void sum(int i)

. +, %, -,
. A+b //variable, value
. S+S1;

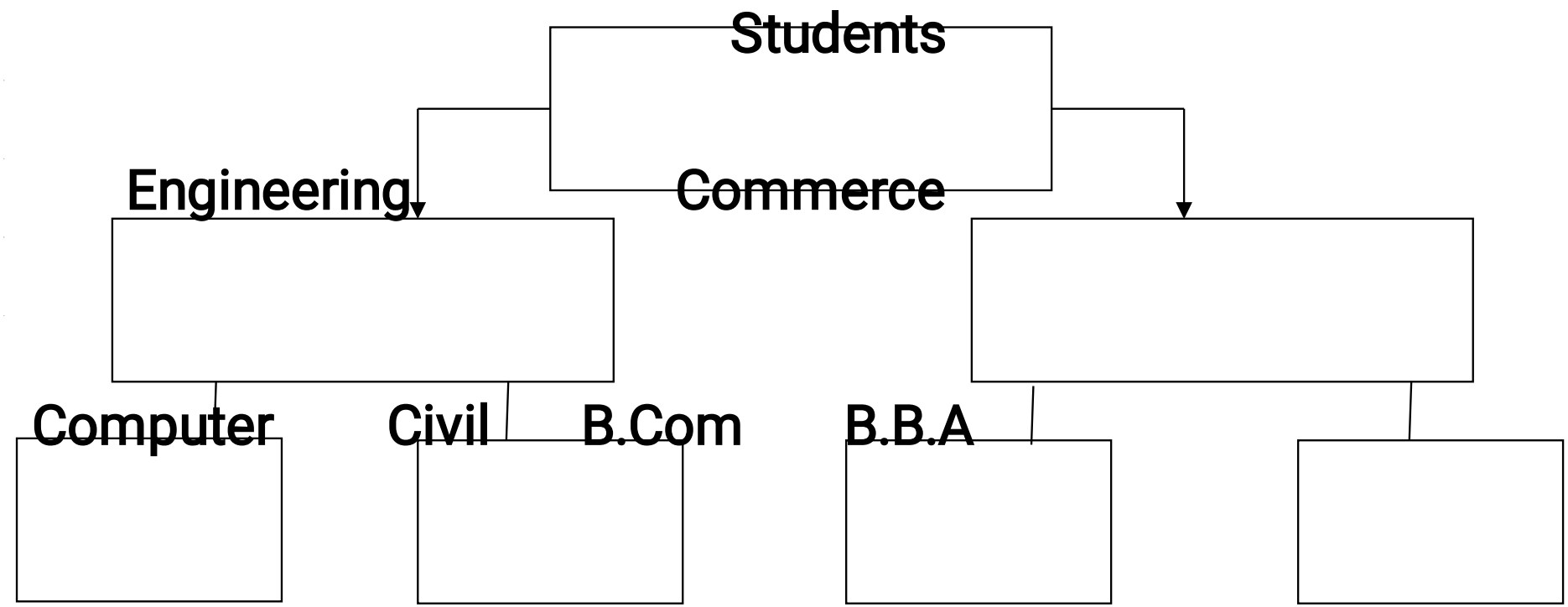
```
Void sum (int j)
{
Cvbfghghghgf;
Fbfghgjhj;
}
Void sum(int I,int j)
{
}
Sum(5,6);
Sum(2);
```

Inheritance :

- Meaning
- Class uses properties of another class (heredity)
- Provides **hierarchical classification** to your program
- **Code reusability**
- Parent (Base) class and Child (Derived) class
- Relationship

```
. Class s1
. {
    Private int a;
      void print()
      {
        xvdfgbf
      }
. };
. Class s2
. {
. }
```

Hierarchical classification :



Reusability :

- “Write once, use many times”
- Can use data and functions of one class in its derived class
- See fig.

Benefits of OOP :

- Saves program development time
- Gives higher security to your code
- Redundant code can be removed using inheritance
- Secure code can be built using data hiding
- Large code can be divided using objects
- System developed under OOP is easily upgraded

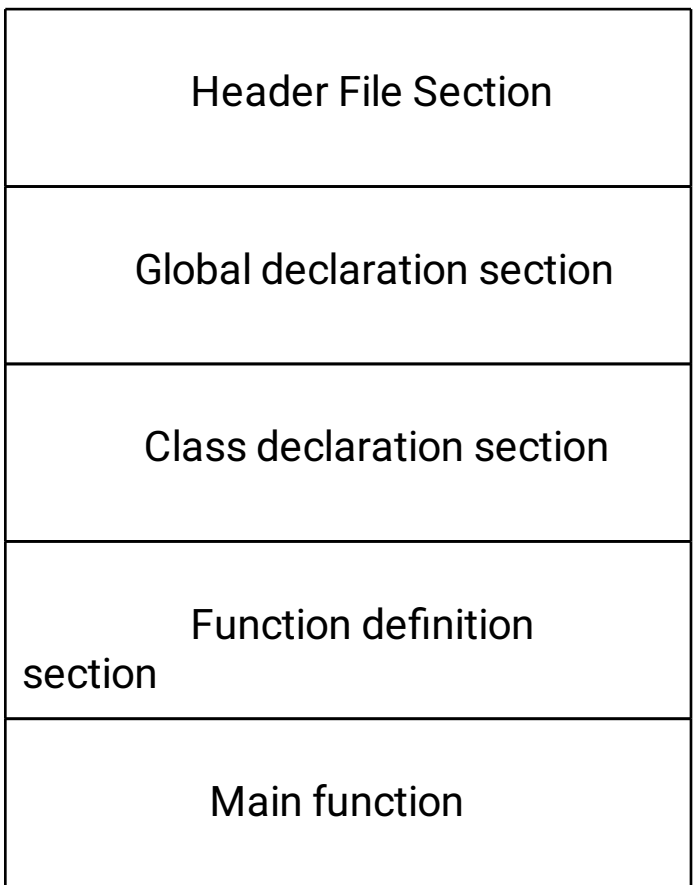
Applications of OOP :

- Real-time applications can be built like stock management, medical software, etc
- CAD-CAM systems
- Object oriented database systems

What is C++ ?

- C++ is an object oriented programming language
- Developed by Bjarne Stroustrup at AT & T Bell Lab, New Jersey, USA in 1980s
- After C language, adding oop concepts to C language, new language was developed and that was C++
- Earlier, it was “**C with classes**” and then after, renamed as C++
- Name derived from increment operator (C++)
- Indicates increment (updation) in C language

Structure (Skeleton) of C++ :



Header file section :

- In C++, <iostream.h> header file is included
- IO stands for Input-Output
- C language header files can also be used in C++ program

Global Declaration Section :

- Global variables are declared
- That can be used throughout the entire program
- Optional section
- Ex : #define PI 3.14, int MAX

Class Declaration Section :

- User can declare class in this section
- It is also optional
- C++ program can also be created without creating class
- We can declare functions and variables in class

Function Definition Section :

- Function declared in class section, can be defined in this section
- Optional section

Main Function :

- Compulsory section
- Entry point for execution of program
- In this section, generally, object of the class are created
- When you run program, OS Calls main function automatically

Sample Program :

```
#include<iostrem.h>       demo.cpp
#include<conio.h>
Void main()          cout-> console output, endl-> end of line
{
   clrscr();
   cout<<“\n Welcome to C++ program”;     // printf(“\n hello”);
   cout<<“Bye”;
   getch();
}
```

Program Using Variable :

```
#include<iostream.h>
#include<conio.h>
Void main()
{
   char c=‘k’;
   cout<<“value of c is :”<<c;
   cout<<“over”;
   getch();
}
```

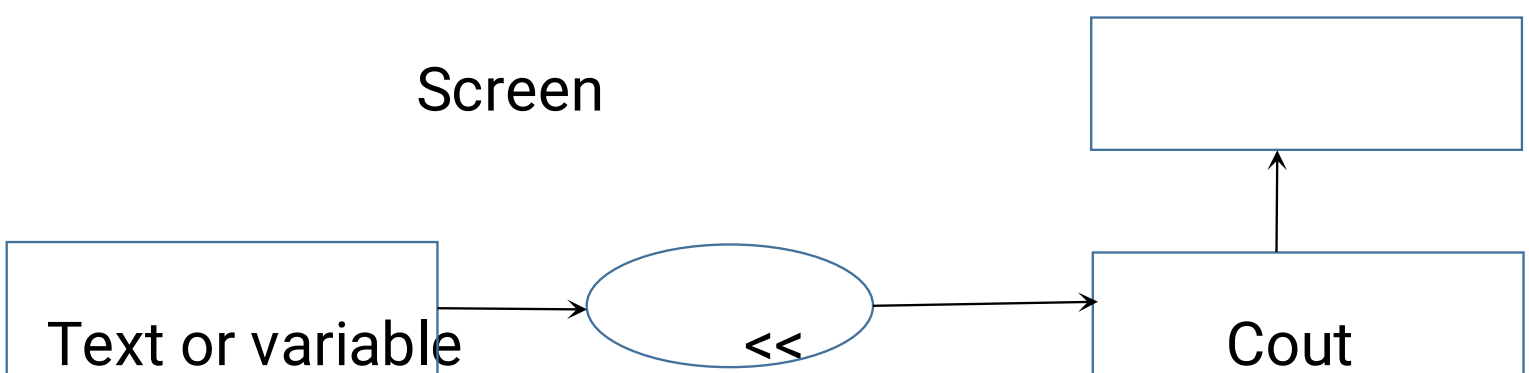
Program Using Class :

```
#include<iostream.h>
class demo
{
   public:
      string name;
      void printname()
      {
      cout<<“Name is :”<<name;
      }
};
```

```
Void main()
{
   demo d;
   d.name=“kamani college”;
   d.printname();
}
```

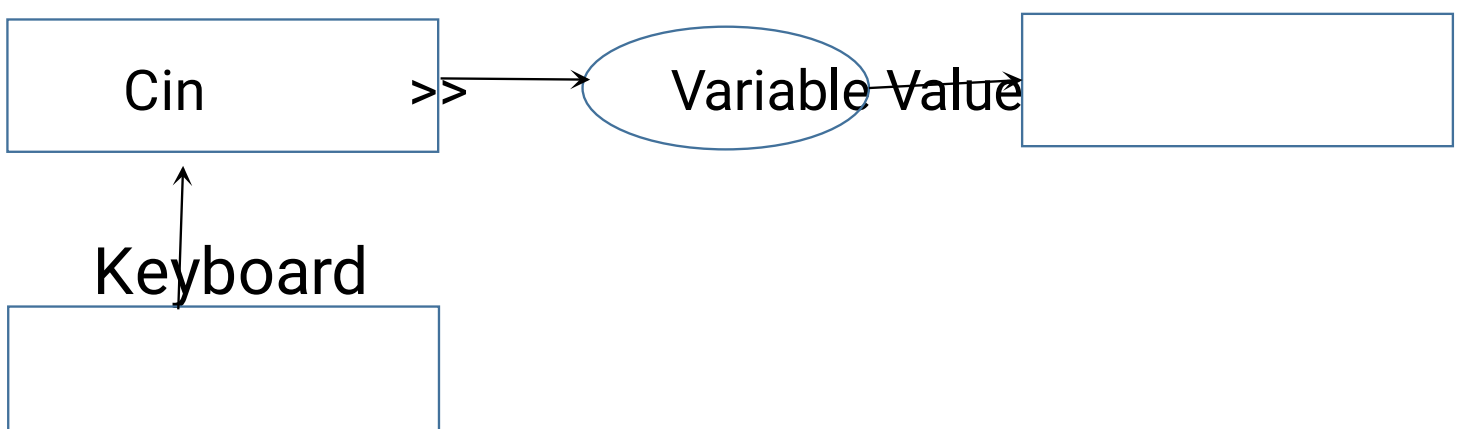
Input/Output Operators :

- The input operator is used to take input from user and the output operator is used to display something on output screen.
- cout statement is used to print output.
- The << operator used with cout is known as **Insertion** operator because it inserts the text or value of variable passed with it.



Input Operator :

- To take some input from keyboard, cin is used
- The symbol is >> that is known as **Extraction** operator.
- It extracts the value entered through the keyboard



Cont....

- >> scans some value through keyboard and it works like…
- cin >> var_name;
- The value you entered is stored in this variable
- We can also give multiple values like,
- cin>>var1>>var2;
- cin>>a>>b>>c>>d>>e>>f;

Simple example :

```
#include<iostream.h>
Void main()
{
   int a,b,c;
   cout<<“Enter a”<<endl;
   cin>>a;  //10
   cout<<“Enter b”<<endl;
   cin>>b;  //20
   c=a+b;   //30
   cout<<“Addition of a and b is”<<c;
}
```

Introduction to Namespace :

- Problem regarding same file name and folder
- Namespace is collection or container of classes
- You can add class, functions, variables etc code in a namespace to create a useful of related code
- You can also create another namespace and add classes, functions, variables with same name of old namespace name without clash.
- Syntax : namespace namespace_name

```
{
   class demo
}
```

Cont….

- To use class or function or both from the namespace, you have to include a statement ,

  using namespace namespace_name;

Ex : Using namespace std;

- Namespace concept will not work on Turbo C++ editor.
- dev C, Visual Studio

Tokens :

- Tokens can be divided into following categories :
- Keywords
- Identifiers
- Constants
- Operators
- Strings

Keywords :

- Reserved identifiers of C++ that have some specific meaning
- You cannot use any keyword name as variable or function or even class and object
- Approx 48-52 keywords are available in C++
- Ex : if, for, case, break, continue, return, class, struct, switch, char, int, float ..

Identifiers :

- It is a name given to variable, function, class, structure, union, object to identify it in a program (class student)
- We can give name to identifiers but with some rules :
  - Keywords cannot be used as identifiers (void getdata())
  - White space is not allowed
  - No any special symbol except underscore (_) is allowed (int $)
  - Digits are allowed but should not start with it (int s1, void sum1())
  - Uppercase and lowercase letters are different (int j,J, void SUM(), void sum())

Constants :

- Constants are fixed values that should not be changed throughout the program execution

- Constants can be of any data type like integer, float, char or string

- #define PI 3.14
- PI++

Operators :

- From students

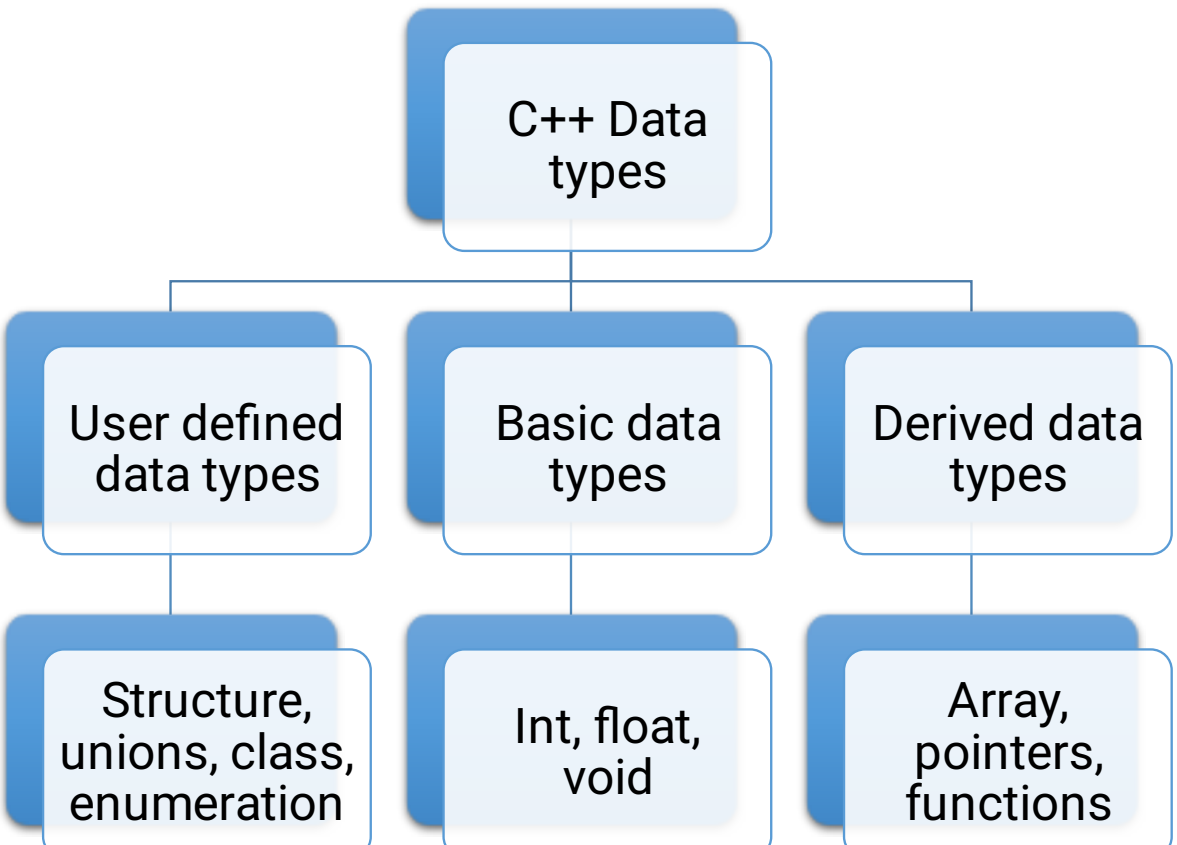
- Struct s1
- {
  Int a; 2
  Float b; 4
  Char c[100]; 1
- };

7

. Union u1
. {
  Int a;
  Float b;
  Char c;
. }

4

Data Types :


C++ Data types
User defined data types
Basic data types
Derived data types
Structure, unions, class, enumeration
Int, float, void
Array, pointers, functions

Enumeration :

- Using enumeration, we can enumerate a list of values
- It assigns 0 to the first value, 1 to second value and so on
- Ex : enum fruits {apple, mango, grapes};
  Here, apple=0, mango=1, grapes=2

  enum flowers {rose, lotus, jasmine=4, sunflower};
  Here, rose=0, lotus=1, jasmine=4, sunflower=5

Pointers : / Reference variables :

- Pointer is a variable that stores address of another variable
- Ex : int a= 10; float *p; p=&a;
- Here a is a variable of int type to store some value and p is a pointer to integer that store address of a.

Symbolic Constants :

- In C++, symbolic constants can be declare in **3 ways** :

1. Using **const** keyword
2. Using **enum** keyword
3. Using **#define** directive #define PI 3.14

For Const, syntax is : const datatype var = value;

Const x=50;

If we do not declare datatype, by default it will take it as **int**

Type Compatibility :

- It refers to whether the types of all variables are compatible to one another or not when written in an expression.
- If in an example, we have 3 variables of different types in an expression, they must be compatible to each other
- Ex : int a =10;

  float b =20;

  double c;

  c=a+b;

Here, it will execute with no error. Type conversion takes place.

Dynamic initialization of variables :

- What is **static** initialization?

int a =10 ;

- What is **dynamic** initialization?

Cin>>a

Operators in C++ :

Operator	Meaning
<<	Insertion operator
>>	Extraction operator
::	Scope resolution operator
::*	Pointer to member declaration operator
->*	Pointer to member access operator
.*	Pointer to member access operator
new	Memory allocation operator
delete	Memory release operator
endl	New line operator
setw	Field width operator

Scope Resolution Operator :

- In program, you can have same variables names in different blocks i.e. inner block and outer block
- The variable of outer block can be accessed by the inner block, the variable declared in outer blocks are known as global variables and the variables declared in the inner block are local variables.
- Thus local and global variables define their scopes regarding how long they are visible

Example :

```
#include<iostream.h>
Int x = 100;    //global variable
Void main()
{
   int x=10;  // local variable
   cout<<x;  //10
   cout<<::x;//100
}
```

Cont…

- You can use scope resolution operator to access the global variable in C++
- Syntax : ::var_nm;
- Int a=99;

```
Void main()
{
   int a=1;
   cout<<“local a is “<<a;   //1
   cout<<“global a is”<<::a;   //99
}
```

Cont…

- The another use of scope resolution operator is in class
- It is used when,
- We have declared a member function in a body of class but,
- We want to give body of the function outside the class.

```
Class demo
{
   public:
      void display();
};
```

Manipulators :

- C++ provides some manipulators used for formatting output
- Only 2 manipulators are :
- endl
- setw
- The endl (end line) is used to insert a new line
- The setw (set width) is used to specify width of variable to be printed in cout.

Example of setw :

```
#include<iostream.h>
#include<iomanip.h>
Void main()
{
   int a =123;
   int b = 12345;
   int c = 1234;
   cout<<“a = “<<setw(5)<<a<<endl;
   cout<<“b=“<<setw(5)<<b<<endl;
   cout<<“c=“<<setw(5)<<c<<endl;
}
```

Type Cast Operators : (Conversion)

- Home Work
- 2 types
  - Implicit (automatic)
  - Explicit (forcefully)

- Float b;
- Int a=(int)b;

Control Structure :

- Also known as **branching** or **decision making statements**

1. If statement
2. If…. Else statement
3. Else….. If ladder (if… else if… else)
4. Nested if statement
5. Switch statement

If Statement :

- Used to execute some code based on a condition
- If condition is true, the code is executed
- **Syntax :** if (a>b)

```
    {
    cout<<“hello”;
  }
    cout<<“hi”;
    cout<<“bye”;
    cout<<“good bye”;
```

If… else Statement :

- Used to execute a block of code if the condition is true, and if the condition is false, the other block of code is executed
- **Syntax :** if(condition)

```
{
   true part
}
else
{
   false part
}
```

Else if Ladder :

- Used to execute a code based on **multiple conditions**
- Each condition is tested one after another
- The block with true condition will be executed

Cont...

- **Syntax :**

```
if(condition)
{
    block1
}
else if(condition)
{
    block2
}
else if(condition)
{
    block3
}
……..
else
{
    block n
}
```

Nested if Statement :

- You can nest multiple if blocks to test the one condition based on another condition
- Syntax :

```
if(condition1)
{
    if(condition2)
    {
        block1
    }
    else
    {
        block2
    }
}
Else
{
    block3
}
```

Switch Statement :

- When we have relatively more conditions, else if ladder or nested if else statement makes the program complex
- In that case, the switch statement proves very handy to make the program
- We can also create **menu driven** like program with switch
- **Syntax** :switch(variable/ expression)

```
{
    case val1:
    statements
    break;

    case val2:
    statements
    break;

    case val3
    statements
    break;
    ……..
    default:
    statements
    break;
}
```

Example :

```
#include<iostream.h>
Void main()
{
   int a,b;
   cout<< “Enter a”;
   cin>>a;  //10
   cout<<“Enter b”;
   cin>>b;  //20
If(a>b)
{
   cout<<“a is big”;
}
Else
{
   cout<<“b is big”;
}
}
```

Example :

```
#include<iostream.h>
Void main()
{
   int a;
   cout<<“enter a”;
   cin>>a;
   if(a>0)
   {
      cout<<“a is positive”;
   }
Else if(a<0)
{
   cout<<“a is negative”;
}
Else
{
   cout<<“a is zero”;
}
}
```

Example :

```
#include<iostream.h>
Void main()
{
   int a=100;
   int b=200;
   if(a==100)
   {
      if(b==200)
      {
         cout<<“value of a is 100 and b is 200”;
      }
}
}
Cout<<“ value of a=“<<a<<endl;
Cout<<“value of b =“<<b;
}
```

Looping Statement :

- Looping statements are used to execute a block of statements repeatedly until a condition is true.
- C++ has 3 loop statements :

1. While loop
2. Do…. While loop
3. For loop

While loop :

- While is the simple loop which just tests a condition and if the condition is true then the block is executed again
- While loop is an **entry-controlled loop** as the condition is checked at the time of entering the block
- **Syntax :** while(condition)

```
{
   block of statements
}
```

Example :

```
#include<iostream.h>
Void main()
{
   int a=1;
   while(a<=5)
   {
      cout<<a<<endl;
      a++;
   }
}
```

Do….. While loop :

- This loop is **exit controlled loop**
- It first executes the block and then tests the condition
- If the condition is true, the block of code is executed again
- It checks the condition at the time of exiting from block, so it is known as exit controlled loop
- Syntax : do

```
{
   block of code
}
while(condition);
```

Example :

```
#include<iostream.h>
Void main()
{
   int i=1;
   do
   {
      cout<<i<<endl;
      i++;
   }
   while(i<=10);
}
```

For loop :

- For loop is simple and very handy
- It is also **entry controlled loop** because the condition is tested before entering the block
- Syntax : for(initialization; condition; increment/decrement)

```
{
   block of code
}
```

- Sequence

Example :

```
#include<iostream.h>
Void main()
{
   cout<<“This is an example of for loop”<<endl;
   for(int i=0;i<5;i++)
   {
      cout<<i<<endl;
   }
   cout<<“out of loop”;
}
```

Functions in C++ :

- Building blocks of C++
- A block of code that performs some specific task
- Used to divide a large and complex program into blocks to become more readable
- Increases the usability of program, reusability
- Function has 3 parts :

1. Function declaration (prototype)
2. Function definition
3. Function call

Cont…

- Declaration specifies the structure of the function
- Tells the compiler, the name of function, number and types of arguments, also return type
- Definition defines what the function will do when it is called
- Function call is the statement which calls the function

- **void display (int , float);** // declaration
- **void display()** **// definition**

```
{
   cout<<“This is definition part”;
}
Display();
```

The main function :

- It is the entry point of the program
- The main function can take arguments that is known as **command-line arguments**
- General form is :
- int main (int numofargs, char *args[])
- Idea of int main() and void main()

```
Int main()
{
    return 0;
}
```

Function Prototype :

- In C++, the function must be declared before it's called
- The function declaration is also known as function prototyping
- It specifies the structure of function
- It tells the compiler about function name, return type and number and types of arguments
- Structure of function prototype :
- **return_type function_name(args_list);**
- The argument list may be empty, example
- **void test();**

Cont…

- The name of the argument is not necessary, but the type is must
- For example,
- void display (int, int, int); will work
- You can write : void display (int a, int b, int c);

Call by reference :

- When the function is called, its arguments passed to the function definition
- In C, the arguments are passed by value, so that, any changes made to the formal arguments will not affect the original values
- But in some cases, if we may need to change the values passed as arguments, so for this, we use pointers and addresses of arguments to call a function by reference
- But in C++, we have the concept of reference variable
- So it will create alias of actual argument of calling function so any changes made on formal arguments will reflact on actual arguments

Cont …

```
Void exchange (int &x, int &y)
{
   int t = a;
   a = b ;
   b = t ;
}
```

Here, x and y are reference variables, so when we call **Exchange (a,b)**

Cont ….

The variables x and y becomes aliases of a and b. so the changes made on x and y will reflect on a and b

```
#include<iostream.h>
#include<conio.h>
Void change(int &x, int &y);
Void main()
{
   int a, b;
   clrscr();
   cout<<“Enter a”;
Cin>>a;
Cout<<“Enter b”;
Cin>>b;
Cout<<“Before swap”;
Cout<<a<<endl<<b;
Change(a,b);
Cout<<“after swap”;
Cout<<a<<endl<<b;
Getch();
}
Void change( int &x, int &y)
{
    x=x+10;
    y=y+10;
}
```

Return by reference :

- As a function is called by reference, a function can also return a reference
- Syntax :

```
return_type &func_name (reference_args)
{
   statements;
}
```

- Function call : func_name() = value;

Example :

```
#include<iostream.h>
#include<conio.h>
Int &min(int &, int &);
Void main()
{
   int a,b;
   clrscr();
   cout<<“enter a”;
   cin>>a;
        cout<<“enter b”;
   cin>>b;
Min(a,b) = 0;
Cout<<“A = “;
Cout<<“B =“;
Getch();
}
 int &min(int &x, int &y)
{
   if(x<y)
      return x;
   else
      return y;
}
```

Inline Functions :

- One of the objective of using functions is to save some memory space which becomes good when a function is likely to be called many times
- Every time a function is called, it takes a lot extra time in executing series of instructions for tasks such as jumping to the function, saving registers, pushing arguments into the stack, returning to calling function
- When a function is **small**, the percentage of execution time may be spent lower

Cont…

- The solution of this problem in C language was
- But problem was that, macros were not actually functions and they cannot detect error during compilation
- **C++** has different solution to this problem
- To eliminate the cost of time, C++ has the concept of small functions called **INLINE FUNCTIONS**
- Inline function is the function that is expanded in line when it is invoked

Cont…

- Syntax : inline return_type fun_nm()
  {
    function body
  }
- Example : inline double cube(double a)
  {
    return (a*a*a);
  }

Cont…

- It is very easy to make a function inline
- All we need to do is to prefix the keyword **inline** to the function definition
- All inline functions must be defined before they are called
- The speed benefits of inline functions diminish as the function grows in size
- At the point of function call, the benefits of inline function may be lost
- Usually inline functions have body of one or two lines

Situations in which inline will not work :

- Function returning values
- Loops, switch, or goto statement
- Function containing static values
- Recursive inline function
- Example : #include<iostreamh>

```
inline float mul(float x, float y)
{
  return (x*y);
}
```

Cont…

```
inline double div (double p, double q)
{
  return (p/q);
}
void main()
{
  float a=12.34, b=32.88;
  cout<<mul(a,b);
  cout<<div(a,b);
}
```

Default Arguments :

- In a function, you can have some or all of your arguments with default values
- These arguments are known as default arguments
- The default arguments are specified at the time of function declaration
- So when compiler sees function prototype it checks for any default arguments and prepare itself for default arguments
- Void sum(int a, int b, int c=0);
- Here third argument has default value 0 which means if the third arg is not passed when function is called, it will be considered as 0

Cont…

- If a function call : sum(10,20);
- The value of a is 10, b will be 20 and c will become 0
- For function call : sum(5,10,15);
- a will be 5, b will be 10and c will be 15
- **Rules :**
- We can have number of default arguments in function
- But, we cannot have default arguments from starting or middle of arg list

Cont…

- void demo(int a, int b=5); //valid
- void demo(int a=10, int b); //invalid
- void demo(int a, int b=10, int c); //invalid
- void demo(int a=10, int b, int c=50); //invalid
- Using default args we can combine similar functions into one
- If we need to create 3 functions; one for adding 2 values, second for adding 3 values and the third for adding 4 values, we can achieve this by making only one function using default argument

Example :

```
#include<iostream.h>
#include<conio.h>
Void sum(int a, int b, int c=0, int d=0);
Void mul(int a, int b, int c=1);
Void main()
{
    clrscr();
    sum(10,20,30,40);
    sum(30,40,50);
    sum(100,200);
    mul(2,3,4);
    mul(4,5);
    getch();
}
```

```
Void sum(int a, int b, int c, int d)
{
    int s=a+b+c+d;
    cout<<“sum = “<<s;
}

Void mul(int a, int b, int c)
{
    int m= a*b*c;
    cout<<“Multiplication is:”<<m;
}
```

Const Arguments :

- If you do not want to allow the function to modify its arguments, you can have constant arguments in your function
- When you declare a function’s argument by const keyword, you will not be able to modify its arguments in function definition
- If you try, you will get compilation error

Example :

```
#include<iostream.h>
#include<conio.h>
Int test(const int a, int b);
Void main()
{
   clrscr();
   cout<<“sum=“<<test(10,20)
   getch();
}
```

```
Int test(const int a, int b)
{
   b=a+b;
   //a=a+b;
   return b;
}
```

Function Overloading :

- Overloading is the concept under Polymorphism
- In Function Overloading, a same function name can be used for performing different tasks
- As we can define different functions with same name, you have to take care of one thing that is the number and type of arguments
- Ina class, if there is more than one function with **same name** but **different types of arguments** then this function is known as overloaded function and this process is known as Function Overloading

Cont…

- For example,
- void test(int x, int y);
- void test (int x, int y, int z);
- void test(int x, float y, char z);
- void test(int a, float b);

Now, Consider the function calls,

- Here,
- test(10,20); //prototype1
- test(1,2,3); //prototype2
- test(1,2.2,'a'); //prototype3
- test(3, 4.5); //prototype4

Working with it :

- When an overloaded function is called, the compiler matches the prototype matching the same number and type of arguments
- If the exact match is not found, the compiler tries to convert the arguments implicitly. For example, int is converted to float, float is converted to double etc
- If the compiler finds multiple matches for a function call, it will give an error message
- Example, void test(int a);

  void test (float b); And call is: test(100);

Cont…

- The compiler will have confusion that which function to call as both of them is applicable for the function call.
- 100 is an integer number and can convert to float also
- If all of the above steps fail, the compiler will try the user-defined conversions to find a suitable match.

Example :

```
#include<iostream.h>
#include<conio.h>
Double area(double r);
Double area(double b, double h);
Void main()
{
    double r,b,h;
    clrscr();
    cout<<“Enter radius”;
    cin>>r;
    cout<<“Area of circle”<<area(r);
    cout<<“Enter b & h”;
    cin>>b>>h;
    cout<<“Area of triangle”<<area(b,h);
    getch();
}
Double area(double r)
{
    double a=3.14*r*r;
    return a;
}
Double area(double b, double h)
{
    double a=0.5*b*h;
    return a;
}
```

Adding C functions in Turbo C++ :

- In C++, we can use C functions also. We should have the knowledge of which header file consists the functions
- For example, to use functions like printf() and scanf(), we have to include stdio.h header file

```
#include<iostream.h>
#include<stdio.h>
#include<conio.h>
Void main()
{
   printf(“hello”);
   cout<<“\n hi”;
   getch();
}
```

Adding UDF into C++ library :

- We can also add our own created function into turbo C++ library.
- Just write function into a file and save it by extension “.h”
- Now include this file into your program, that’s it.
- Steps:
  - Write function in a file
  - Save File in include directory
  - C:\Turboc3\INCLUDE
  - Save it like “test.h”
  - Now create any program and write #include<test.h>
  - Save, Compile and execute it