C++ and Java Programming
18K3CS04

M.IDA ROSE

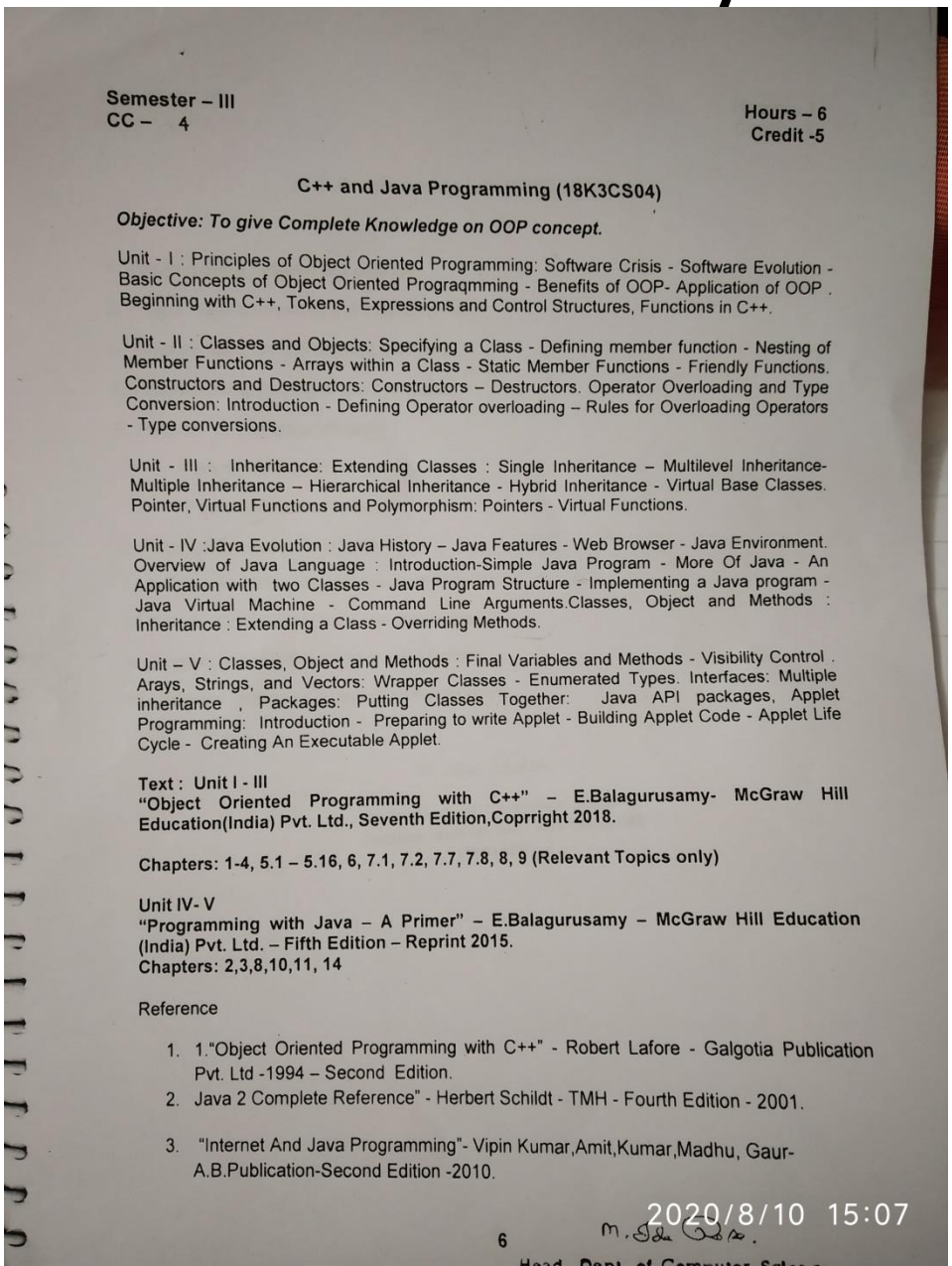
Assistant Professor

Department of Computer Science

K.. N. Government Arts College for Women(Autonomous)

Thanjavur – 613 007

Syllabus



Semester – III
CC – 4

Hours – 6
Credit -5

C++ and Java Programming (18K3CS04)

Objective: To give Complete Knowledge on OOP concept.

Unit - I : Principles of Object Oriented Programming: Software Crisis - Software Evolution - Basic Concepts of Object Oriented Prograqmming - Benefits of OOP- Application of OOP . Beginning with C++, Tokens, Expressions and Control Structures, Functions in C++.

Unit - II : Classes and Objects: Specifying a Class - Defining member function - Nesting of Member Functions - Arrays within a Class - Static Member Functions - Friendly Functions. Constructors and Destructors: Constructors – Destructors. Operator Overloading and Type Conversion: Introduction - Defining Operator overloading – Rules for Overloading Operators - Type conversions.

Unit - III : Inheritance: Extending Classes : Single Inheritance – Multilevel Inheritance- Multiple Inheritance – Hierarchical Inheritance - Hybrid Inheritance - Virtual Base Classes. Pointer, Virtual Functions and Polymorphism: Pointers - Virtual Functions.

Unit - IV :Java Evolution : Java History – Java Features - Web Browser - Java Environment. Overview of Java Language : Introduction-Simple Java Program - More Of Java - An Application with two Classes - Java Program Structure - Implementing a Java program - Java Virtual Machine - Command Line Arguments.Classes, Object and Methods : Inheritance : Extending a Class - Overriding Methods.

Unit – V : Classes, Object and Methods : Final Variables and Methods - Visibility Control . Arays, Strings, and Vectors: Wrapper Classes - Enumerated Types. Interfaces: Multiple inheritance , Packages: Putting Classes Together: Java API packages, Applet Programming: Introduction - Preparing to write Applet - Building Applet Code - Applet Life Cycle - Creating An Executable Applet.

Text : Unit I - III
"Object Oriented Programming with C++" – E.Balagurusamy- McGraw Hill Education(India) Pvt. Ltd., Seventh Edition,Coprright 2018.

Chapters: 1-4, 5.1 – 5.16, 6, 7.1, 7.2, 7.7, 7.8, 8, 9 (Relevant Topics only)

Unit IV- V
"Programming with Java – A Primer" – E.Balagurusamy – McGraw Hill Education (India) Pvt. Ltd. – Fifth Edition – Reprint 2015.
Chapters: 2,3,8,10,11, 14

Reference

1. 1."Object Oriented Programming with C++" - Robert Lafore - Galgotia Publication Pvt. Ltd -1994 – Second Edition.
2. Java 2 Complete Reference" - Herbert Schildt - TMH - Fourth Edition - 2001.
3. "Internet And Java Programming"- Vipin Kumar,Amit,Kumar,Madhu, Gaur- A.B.Publication-Second Edition -2010.

2020/8/10 15:07

6

- Unit I – Chapters: 1, 2, 3, 4
- Unit II – Chapters: 5.1-5.16, 6, 7.1-7.2
- Unit III – Chapters: 8, 9

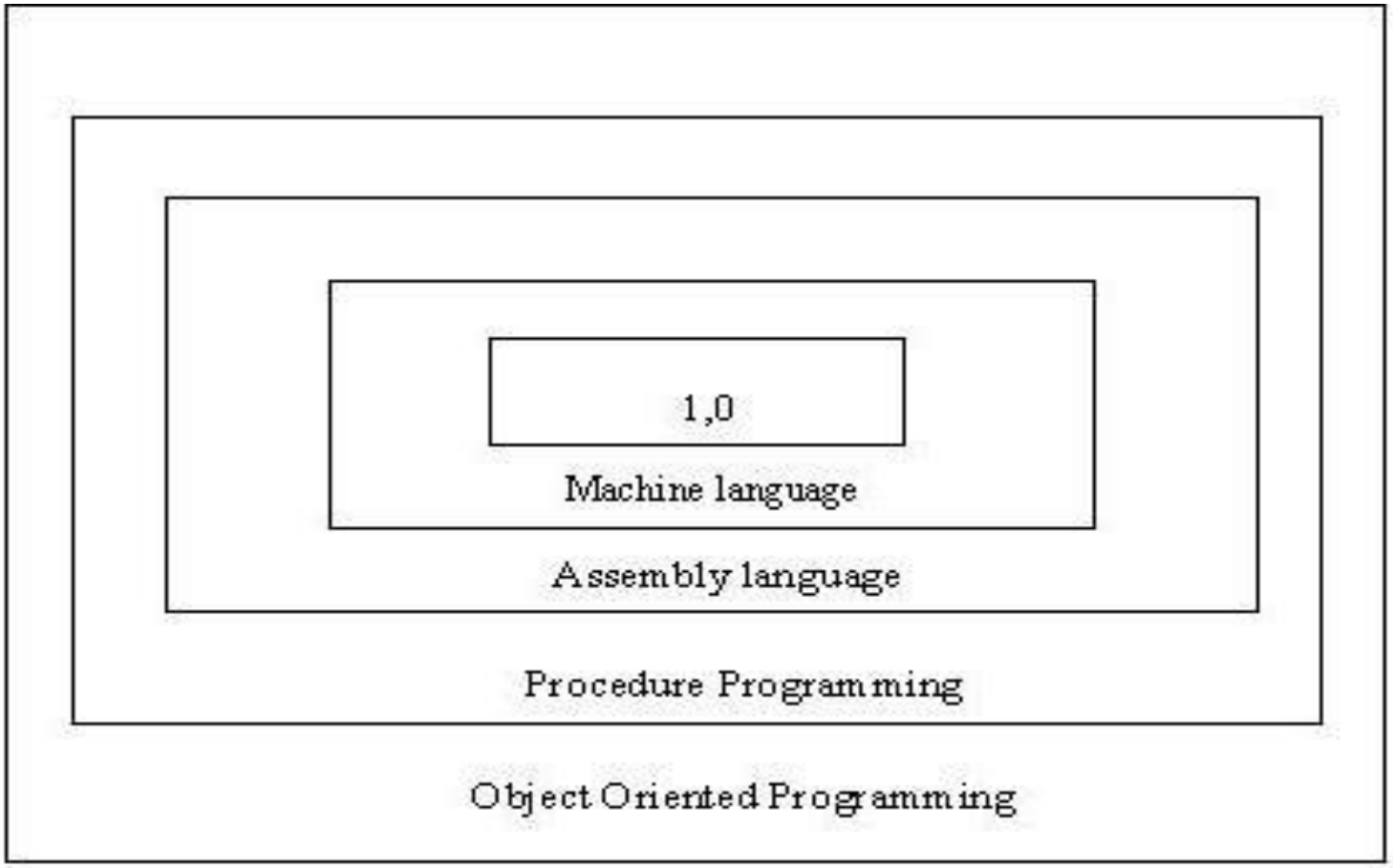
Chapter 1

Principles of Object Oriented Programming

Software Crisis

- “Software Crisis” in procedural programming:
  - Too many modules
  - Too many functions
  - Too many variables
  - …. An expensive mess
- Better organization of the code
- Smaller code
- Reuse of code
- Easier design, analysis and implementation
- User vs. Programmer

Software Evolution



A Look at Procedure – Oriented Programming

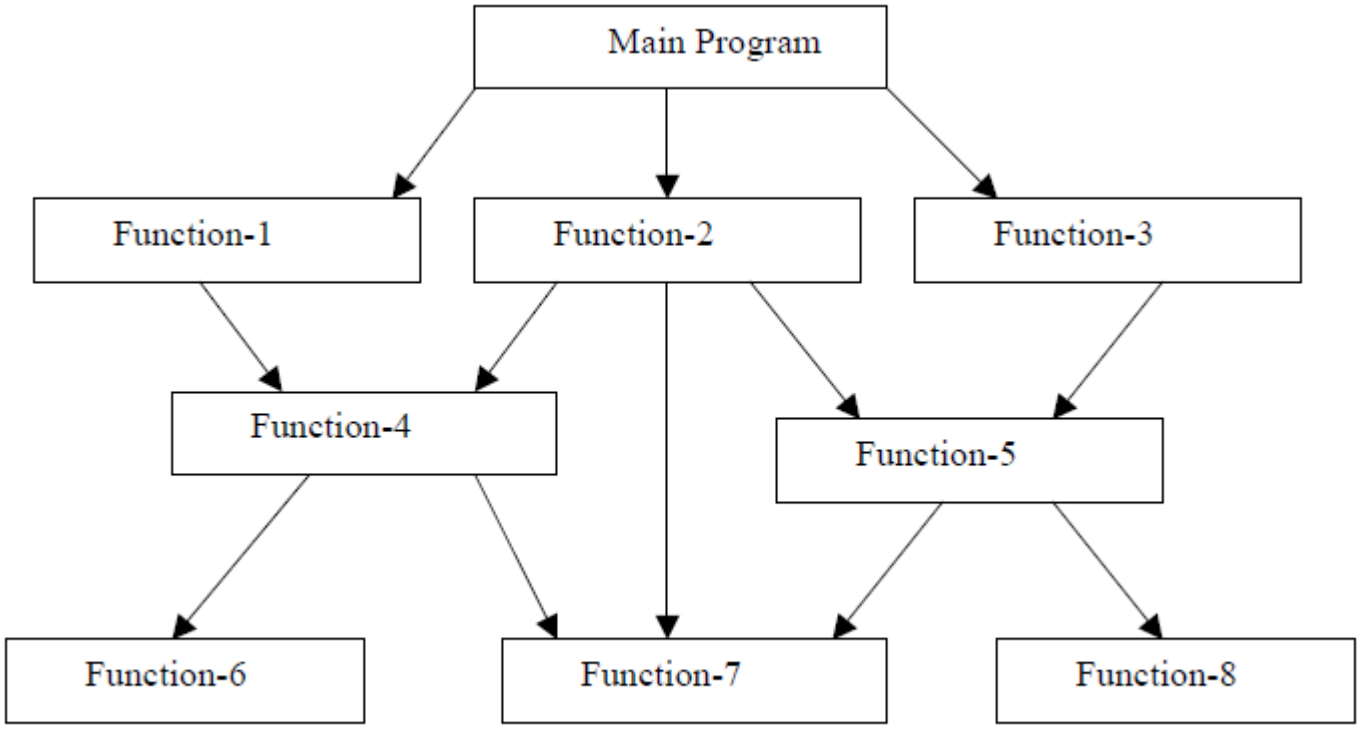
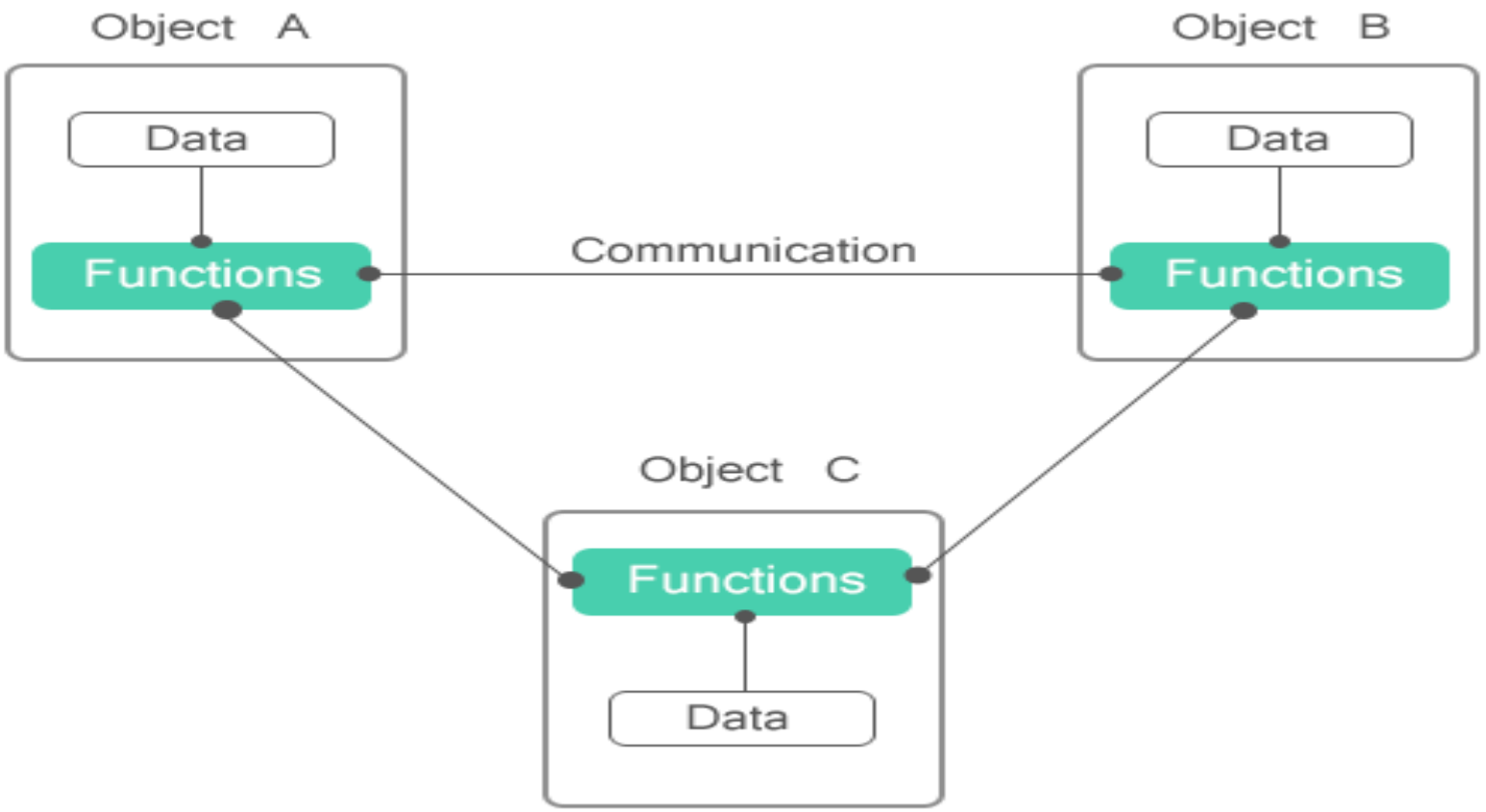


Fig. 1.2 Typical structure of procedural oriented programs

Object – Oriented Programming Paradigm



Basic Concepts of Object – Oriented Programming

- Objects
- Classes
- Data Abstraction and Encapsulation
- Inheritance
- Polymorphism
- Dynamic Binding
- Message Passing

Basic Concepts of OOP

1.Object:

Objects are the basic run – time entities in an object – oriented system.

2.Classes:

We just mentioned that objects contain data, and code to manipulate that data.

Basic Concepts of OOP

3.Data Abstract and Encapsulation:

The wrapping up of data and function into a single unit (called class) is known as encapsulation. Data encapsulation is the most striking feature of a class.

4.Inheritance:

Inheritance is the process by which objects of one class acquire the properties of objects of another class.

Basic Concepts of OOP

5. Polymorphism:

   Polymorphism is another important OOP concept.

6. Dynamic Binding:

   Binding refers to the linking of a procedure call to the code to be executed in response to the call.

Basic Concepts of OOP

7. Message Passing:

An object – oriented program consists of a set of objects that communicate with each other.

Benefits of OOP

- Through inheritance, we can eliminate redundant code and extend the use of existing classes.
- Standard working modules that communicate with one another.
- Data hiding helps the programmer to build secure programs that cannot be invaded by code in other parts of the program.
- It is possible to have multiple instances of an objects to co-exist without any interference.
- It is possible to map objects in the problem domain to those objects in the program.
- Ease to partition that work in a project based on objects.

Object–Oriented Languages

Object-Oriented Languages

- The focus of OOP languages is not on structure, but on *modeling data*.
- Programmers code using "blueprints" of data models called *classes*.
- Examples of OOP languages include C++, Visual Basic.NET and Java.

Object–Oriented Languages

- The focus of OOP languages in is not on structure, but on modeling data.
- Programmers code using “blueprints” of data models called classes.
- Examples of OOP languages include C++, Visual Basic.NET and Java.

Applications of OOP

- The promising areas of application of OOP include:
- Real-time system
- Simulation and modeling
- Object oriented data bases
- Hypertext, Hypermedia
- AI and expert systems
- Neural Networks and parallel programming
- Decision support and office automation systems
- CIM/CAM/CAD systems

Chapter 2

Beginning with C++

Applications of C++

- To create hierarchy-related objects, build special object-oriented libraries
- To map the real-world problem
- Easy maintainable and expandable programs. Easy to implement new features to the existing structure.

Difference between Procedure Oriented Programming and Object Oriented Programming

Procedure oriented programming	Object Oriented programming
Emphasis is on doing things (algorithms) Large Programs are divided into smaller programs known as functions. Most of the functions share global data. Data move openly around the system from function to function. Functions transform data from one form to another. Employs top down approach in program design.	Emphasis is on data rather than on procedure Programs are divided into objects Data structures are designed in such a way that it characterizes the object Functions that operate on data are ties together in a data structure called class Data is hidden and cannot be accesses by external functions. Objects may communicate to each other with the help of functions. New data and functions can be easily added whenever necessary Follows bottom-up approach.

Input and Output Operators

- In C++, input and output (I/O) operators are used to take input and display output. The operator used for taking the input is known as the **extraction** or get **from operator** (>>), while the operator used for displaying the output is known as the **insertion** or **put to operator** (<<).

Input Operator

- The input operator, commonly known as the extraction operator (>>), is used with the standard input stream, cin. As stated earlier, cin treats data as a stream of characters. These characters flow from cin to the program through the input operator. The input operator works on two operands, namely, the c in stream on its left and a variable on its right. Thus, the input operator takes (extracts) the value through cin and stores it in the variable.
-
- To understand the concept of an input operator, consider this example.
-
- A program to demonstrate the working of an input operator.
-
- #include<iostream>
- using namespace, std;
-
- int main ()
- {
- int a;
- cin>>a;
- a = a+1;
- return 0;
- }
- In this example, the statement cin>> a takes an input from the user and stores it in the variable a.

Output Operator

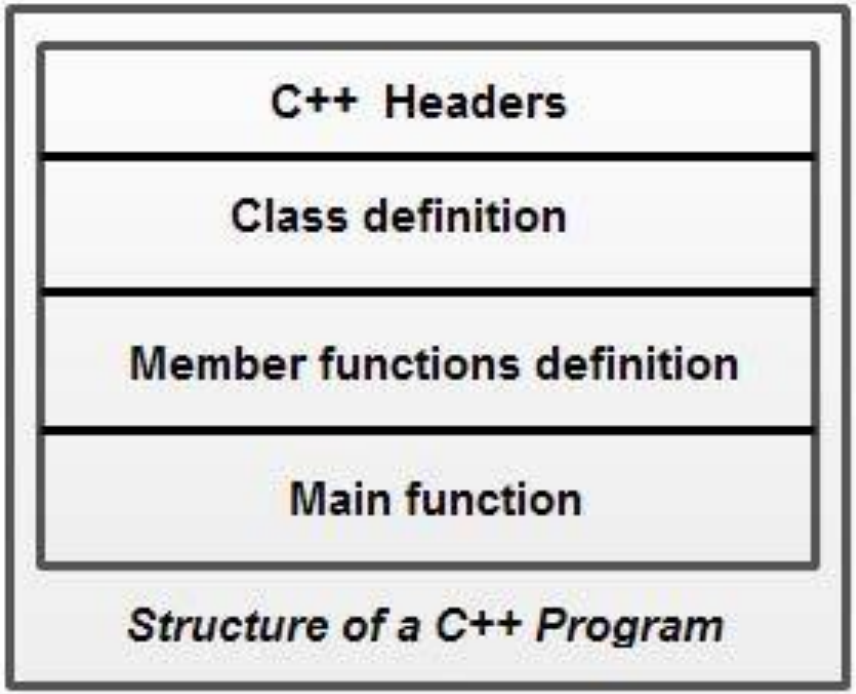
- The output operator, commonly known as the insertion operator (<<), is used. The standard output stream cout Like cin, cout also treats data as a stream of characters. These characters flow from the program to cout through the output operator. The output operator works on two operands, namely, the cout stream on its left and the expression to be displayed on its right. The output operator directs (inserts) the value to cout.
-
- To understand the concept of output operator, consider this example.
-
- A program to demonstrate the working of an output operator.
-
- #include<iostream>
- using namespace std;
- int main ()
- {
- int a;
- cin>>a;
- a=a+1;
- cout<<a;
- return 0;
- }
-
- This example is similar to Example 1. The only difference is that the value of the variable a is displayed through the instruction cout << a .

Cascading the Inputand Output Operator

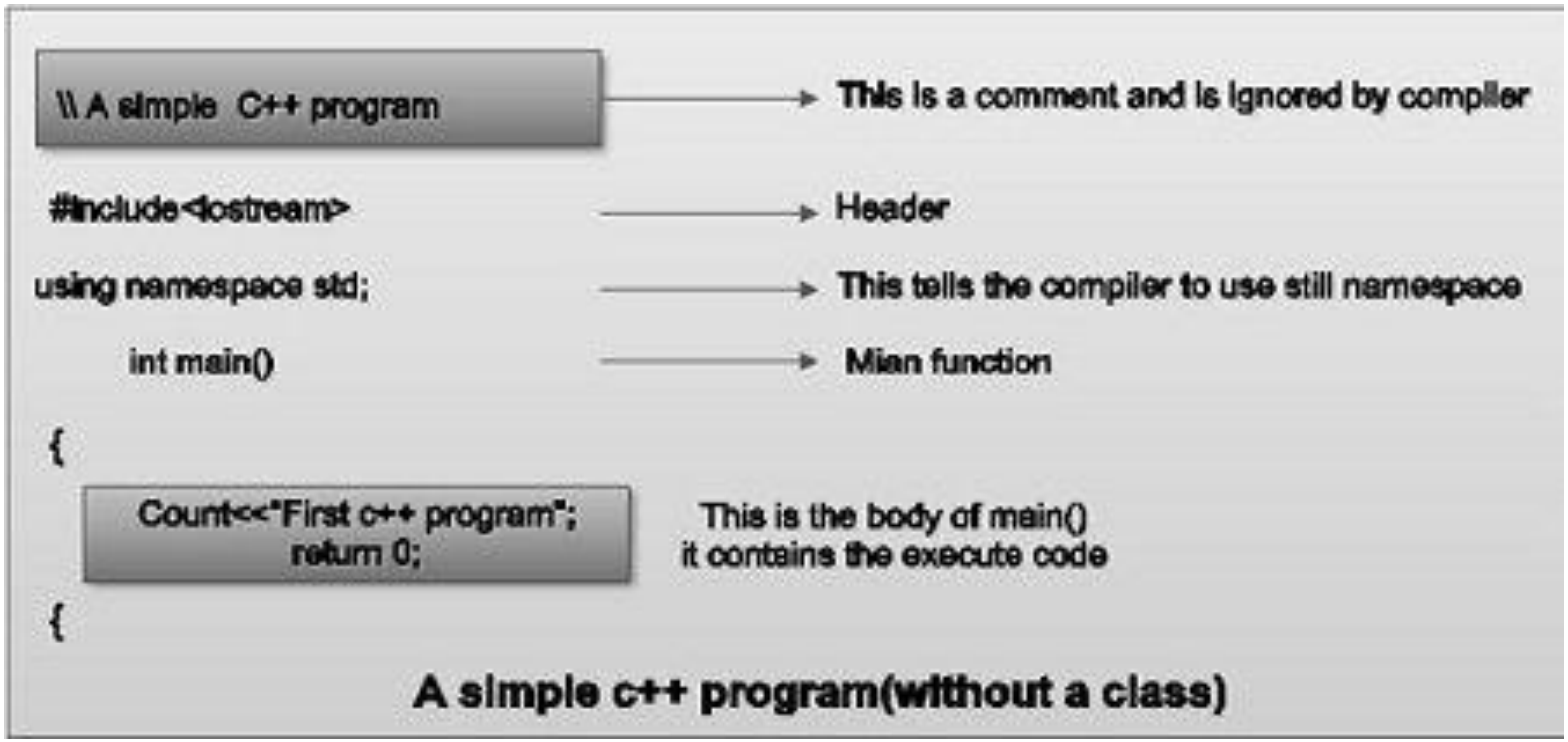
- The cascading of the input and output operators refers to the consecutive occurrence of input or output operators in a single statement.
- To understand the concept of cascading of the input/output operator, consider these examples.
-
- A program without cascading of the input/output operator.
-
- #include<iostream>
- using namespace std;
- int main ()
- {
- int a, b;
- cin>>a;
- cin>>b;
- cout<<"The value of a is
- cout<<a;
- cout<<"The value of b is
- cout<<b;
- return 0;
- }
-
- In this example, all cin and cout statements use separate input and output operators respectively However, these statements can be combined by cascading the input and output operators accordingly as shown in this example.

- A program with cascading of the input/output operator
-
- #include<iostream>
- using namespace std;
- int main ()
- {
- int a, b;
- cin>>a>>b;
- Cout<<"The value of b is : "<<b;
- cout<<"The value of a is ***"<<a;***
- return 0;
- }
-
- In this example, the cascaded input operators wait for the user to input two values and the cascaded output operator first displays the message The value of a is: and then displays the value stored in a. Similar is the case for the next statement.
- It can be observed that cascading of the input/output operator improves the readability and reduces the size of the program.

Structure of C++ Program



Example for C++ Program



Chapter 3

Tokens, Expressions and Control Structures

Tokens

- Keywords
- Identifiers
- Constants
- Strings
- Operators
- The smallest individual units in a program are known as tokens.
- Including white space and syntax of the language.

Keywords

- Definition:
  - The words which are explicitly reserved for identifiers and cannot be used as names for the program variables or other user-defined program elements.
- Examples:
  - double
  - continue
  - private
  - typedef
  - if
  - int
  - long
  - for
  - default

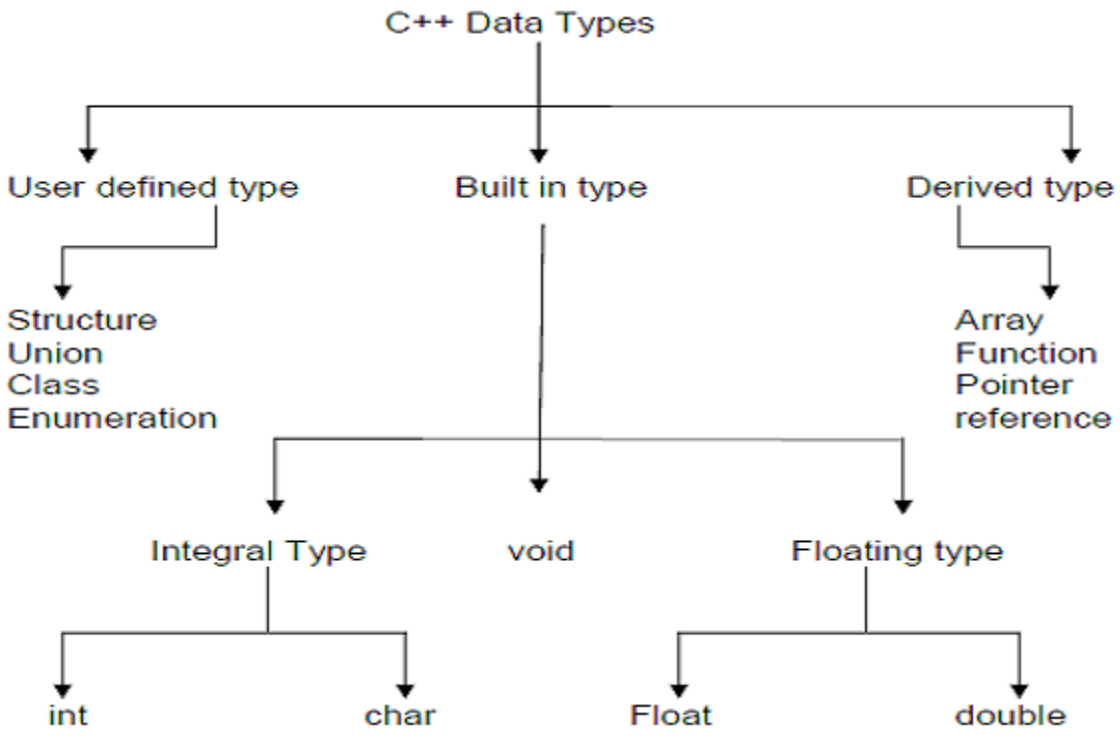
Identifiers and Constants

- Identifiers
  - Definition:
    - Refer to the names of variables, functions, arrays, classes, etc. created by the programmer.
  - Rules
    - Only alpha numeric characters, digits and underscrores are permitted.
    - The name cannot start with a digit.
    - Uppercase and lowercase letters are distinct.
    - A declared keyword cannot be used as a variable name.
    - 32 characters are allowed.

Identifiers and Constants

- Constants:
  - Definition:
    - Refer to fixed values that do not change during the execution of a program.
  - Examples:
    - 123 //decimal integer
    - 12.34 // floating point integer
    - 037 //octal integer
    - 0X2 //hexadecimal integer
    - “C++” // string constant
    - ‘A’ //character constant
    - L’ab’ //wide-character constant

Basic Data Types



Size and Range of C++ basic data types

Data Type	Range	Bytes	Format
signed char	-128 to + 127	1	%c
unsigned char	0 to 255	1	%c
short signed int	-32768 to +32767	2	%d
short unsigned int	0 to 65535	2	%u
signed int	-32768 to +32767	2	%d
unsigned int	0 to 65535	2	%u
long signed int	-2147483648 to +2147483647	4	%ld
long unsigned int	0 to 4294967295	4	%lu
float	-3.4e38 to +3.4e38	4	%f
double	-1.7e308 to +1.7e308	8	%lf
long double	-1.7e4932 to +1.7e4932	10	%Lf

Note: The sizes and ranges of int, short and long are compiler dependent. Sizes in this figure are for 16-bit compiler.

User-Defined Data Types

- Structures
- Unions
- Classes
- Enumerated Data type

User-Defined Data Types

- Various user-defined data types provided by C++ are *structures, unions, enumerations* and *classes.*
- **Structure, Union and Classes:** Structure and union are the significant features of C language. Structure and union provide a way to group similar or dissimilar data types referred to by a single name. However, C++ has extended the concept of structure and union by incorporating some new features in these data types to support object -oriented programming.
- C++ offers a new user-defined data type known as class, which forms the basis of object-oriented programming. A class acts as a template which defines the data and functions that are included in an object of a class. Classes are declared using the keyword class. Once a class has been declared, its object can be easily created.

Structures

- While arrays are used to group together similar type data elements, structures are used for grouping together elements with dissimilar types.
- General format:
  - struct name
  - {
    - datatype member1;
    - datatype member2;
    - …..
    - …
  - }
- Example:
  - struct book
  - {
    - char title[25];
    - char author[25];
    - int pages;
    - float price;
  - };
  - struct book1, book2, book3;

  - book1.pages = 400;
  - book1.price = 525;

Union

- Conceptually both unions and structures are same.
- But the difference is :
  - the size of the structure is equal to the sum of the sizes of individual member types.
  - The size of a union is equal to the size of its largest member element.
- Example:
  - union result
  - {
    - int marks;
    - char grade;
    - float percent;
  - };
- Here size the union occupy four bytes in memory as its largest member element is the floating type variable percent.

enum

- **Enumeration:** An enumeration is a set of named integer constants that specify all the permissible values that can be assigned to enumeration variables. These set of permissible values are known as enumerators. For example, consider this statement.
- **enum country {US, UN, India, China}; // declaring an**
- **// enum type**
- In this statement, an enumeration data-type country (country is a tag name) , consisting of enumerators US, UN and so on, is declared. Note that these enumerators represent integer values, so any arithmetic operation can be performed on them.
- By default, the first enumerator in the enumeration data type is assigned the value zero. The value of subsequent enumerators is one greater than the value of previous enumerator. Hence, the value of US is 0, value of UN is 1 and so on. However, these default integer values can be overridden by assigning values explicitly to the enumerators
- as shown here.
- **enum country {US, UN=3, India, china} ;**
- In this declaration, the value of US is O by default, the value of UN is 3, India is 4 and soon.

enum

- Once an enum type is declared, its variables can be declared using this statement.
- **country countryl, country2;**
- These variables countryl, country2 can be assigned any of the values specified in enum declaration only. For example, consider these statements.
- **countryl India; // valid**
- **country2 Japan; // invalid**
- Though the enumerations are treated as integers internally in C++, the compiler issues a warning, if an int value is assigned to an enum type. For example, consider these statements.
- **Country1 = 3; //warning**
- **Country1 = UN; / /valid**
- **Country1 = (country) 3; / /valid**
- C++ also allows creating special type of enums known as **anonymous enums,** that is, enums without using tag name as shown in this statement.
- **enum {US, UN=3, India, China};**
- The enumerators of an anonymous enum can be used directly in the program as shown here.
- **int count = US;**

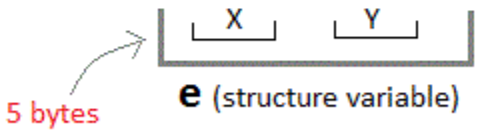
The typedef Keyword

- C++ provides a typedef feature that allows to define new data type names for existing data types that may be built-in, derived or user-defined data types. Once the new name has been defined, variables can be declared using this new name. For example, consider this declaration.
- **typedef int integer;**
- In this declaration, a new name integer is given to the data type into This new name now can be used to declare integer variables as shown here.
- **integer i, j, k;**
- Note that the typedef is used in a program to contribute to the development of a clearer program. Moreover, it also helps in making machine-dependent programs more portable.

Structures and Unions

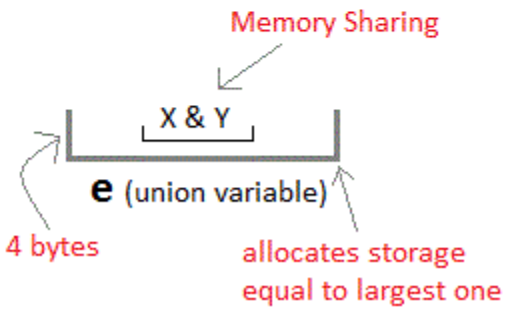
Structure

```
struct Emp
{
 char X ;     // size 1 byte
 float Y ;    // size 4 byte
} e ;
```



Unions

```
union Emp
{
 char X ;
 float Y ;
} e ;
```



Difference between Structure and Union

	STRUCTURE	UNION
Keyword	The keyword **struct** is used to define a structure	The keyword **union** is used to define a union.
Size	When a variable is associated with a structure, the compiler allocates the memory for each member. The size of structure is **greater than or equal to the sum of sizes of its members.**	when a variable is associated with a union, the compiler allocates the memory by considering the size of the largest memory. So, size of **union is equal to the size of largest member.**
Memory	Each member within a structure is assigned unique storage area of location.	Memory allocated is shared by individual members of union.
Value Altering	Altering the value of a member will not affect other members of the structure.	Altering the value of any of the member will alter other member values.
Accessing members	Individual member can be accessed at a time.	Only one member can be accessed at a time.
Initialization of Members	Several members of a structure can initialize at once.	Only the first member of a union can be initialized.

Enumerated Data Type

Enumerated Data Type (**enum**)

An Enumerated data type consists of an ordered set of distinct constant values defined in a data type in a program. The format of **en-um** is:-

```
enum name
{
value-1,value-2,value-3,……..,value-4;
};
```

where., name is the name of the enumerated data type, also known as tag and value-1,value-2,value-3,……..,value-n are values that variable of type **name** can take.

Enumerated data types - Example

Using Enumerated Types

Comparisons

Use relational operators, compare in the same manner as integers:

- `if ( MONDAY < FRIDAY )`
- `if ( MONDAY == 0 )`
- `for (index = MONDAY; index <= FRIDAY; index ++) { ... }`

Math Operations to change

Example 3

```
1 Day today, tomorrow;
2 today = SUNDAY:
3 tomorrow = today + 1;
```

Stepping through an array

Example 4

```
1 // Get the total sales for each day.
2 for (index = MONDAY; index <= FRIDAY; index++)
3 {
4     cout << "Enter the sales for day "
5          << index << ": ";
6     cin >> sales[index];
7 }
```

Storage Classes

Storage Class	Keyword	Lifetime	Visibility	Initial Value
Automatic	auto	Function Block	Local	Garbage
External	extern	Whole Program	Global	Zero
Static	static	Whole Program	Local	Zero
Register	register	Function Block	Local	Garbage
Mutable	mutable	Class	Local	Garbage

Derived Data Types

- Arrays
  - Example:
    - char string[3]=“xyz”;
    - Char string[4]=“abc”; //null is allowed.
- Functions
- Pointers
  - char *const ptr1=“GOOD”; // constant pointer
  - int const *ptr2=&m; //pointer to a constant
  - const char *const cp=“xyz”; //constant pointer to string.

Type Compatibility

- C++ is very strict with regard to type compatibility to C.
- Short int, long int
- Unsigned char, char, signed char
- Int is not compatibile with char
- Function overlaoding

Type Compatibility

- sizeof()
- It returns size of a variable.
- Example:
  - Sizeof(‘x’); //if x is integer, then it will return 1.
  - sizeof(char);

Type compatibility

- The sizeof is a keyword but it is compile time operator that determine the size, in byte, of a variable or data type.
- It can be used to get the size of classes, structure, union and any other user defined data type.
- Syntax:
  - sizeof(datatype)
- Ex:
  - sizeof(x) or sizeof(int)
- sizeof operator will return integer value.

Declaration of Variables

- All variables must be declared before they are used in executable statements.
- Example:
  - Int a;
  - Cin>>a;
  - Cin>>b; //error: variable not declared.
  - Declaration syntax error.
  - For (int i=0; i<10; i++)
  - Int I;
  - For (i=0; i<10; i++)

Declaration of variable

- Syntax:
  - Datatype variablename;
- Example:
  - int a; // a is an integer variable.
  - char x[5]; // x is a character variable; it can store 5 characters. Its size is 5 bytes.
- Syntax:
  - Datatype var1, var2, var3;
- Example:
  - Int a, b, c;

Dynamic Initialization of Variables

- C++ permits initialization of the variable at run time.
- Example:
  - ……
  - ……
  - ……int n=strlen(string);
  - ……
  - ……float area= 3.14 *rad*rad;

Reference Variables

- A reference variable provides an alias (alternative name) for a previously defined variable.
- Syntax:
  - Data-type &reference-name = variable-name
- Example:
  - Float total =100;
  - Float &sum = total;

Operators in C++

- :: scope resolution operator
- ::* pointer-to-member declarator
- ->* pointer-to-member operator
- .* pointer-to-member operator
- delete memory release operator
- endl line feed operator
- new memory allocation operator
- setw field width operator

Scope Resolution Operator

- A variable declared inside a block is said to be local to that block.
- Example:
- ……
- {
- …….int x=10;
- ……
- }
- ……..
- {
- ……
- ……. int x=1;
- ……..
- }

Member Dereferencing Operators

- ::* to declare a pointer to a member of a class
- .* to access a member using object name and a pointer to that member
- ->* to access a member using a pointer to the object and a pointer to that member

Memory Management Operators

- **new**
- Syntax:
  - Pointer-variable =new data-type;
  - Pointer-variable =new data-type(value);
  - Pointer-variable =new data-type[size];
- Example:
  - Array-ptr= new int;
- **delete**
- Syntax:
  - delete pointer-variable;
  - delete [size]pointer-variable;
- Example:
  - delete p;
  - Delete []p;

Manipulators

- Defintion:
  - Manipulators are operators that are used to format the data display.
  - endl \n – new line
  - setw \t – width space
  - setprecision()
- Example:
  - Cout<<“m=“<<m<<endl;
  - Count<<“n=“<<n<<endl;
  - M=2597
  - N=14

Use of manipulators

```
#include<iostream.h>
#include<conio.h>
Using namespace std;
int main()
{
      int Basic=950,Allowance = 95, Total=1045;
      cout<<setw(10)<<“Basic=“<<setw(10)<<Basic<<endl;
      cout<<setw(10)<<“Allowance=“<<setw(10)<<Allowance<<endl;
      cout<<setw(10)<<“Total=“<<setw(10)<<Total<<endl;

      return 0;
}
```

Output:

```
            Basic=          950
            Allowance=    95
            Total=          1045
```

Note: character strings are also right-justified.

Setprecision()

- control the precision of floating point numbers appearing in the output.
  - 10.5
  - 205.7
  - 1050.2

Our own maninpulators

```
#include<iostream.h>
ostream & symbol (ostream &output)
{
   return output <<“\tRs.”;
}
```

Type Cast Operator

- Usage:
  - Explicit type conversion of variables or expressions.
- Implicit conversion : example: Int I; i=10.2; ➔i=10
- Type cast conversion:
  - syntax:
    - type-name expression
  - Example:
    - Average = sum/float (i);

Explicit Type Casting

```
#include<iostream.h>
#include<conio.h>
Using namespace std;
Int main()
{
      int intvar=25;
      float floatvar=35.87;
      cout<<“integer variable=“<<intvar;
      cout<<“\n float variable=“<<floatvar;

      cout<<\n float variable=“<<float(intvar);                    //float(25)
      cout<<“\n integer variable=“<<int(floatvar);                          //int(35.87)

      return 0;
}
```

Output:
Integer variable = 25
Float variable = 35.87
Integer variable = 25
Float variable = 35

New cast operators in ANSI C++

- const_cast
- static_cast
- dynamic_cast
- reinterpret_cast

Expressions and their Types

- Definition:
  - An expression is a combination of operators, constants and variables arranged as per the rules of the language.
  - It may also include function calls which return values.
  - An expression may consists of one or more operands, and zero or more operators to produce a value.

7 types of expressions:

- Constant expressions
- Integral expressions
- Float expressions
- Pointer expressions
- Relational expressions
- Logical expressions
- Bitwise expressions

Constant expressions

- Constant expressions consists of only constant values.
- Examples:
  - 15
  - 20.57
  - ‘x’
  - 20+5/2.0

Integral expressions

- Integral expressions are those which produce integer results after implementing all the automatic and explicit type conversions.
- Example:
  - m
  - m * n - 5
  - m * ’x’
  - 5 + int(2.0)

Float expressions

- Float expresssions are tho0se which, after all conversions, produce floating-point results.
- Examples:
  - x + y;
  - x * y / 10
  - 5 + float(10)
  - 10.75

Pointer Expressions

- Pointer expressions produces address values.
- Example:
  - &m
  - Ptr
  - Ptr +1
  - “xyz”

Relational expressions

- Relational expressions yield results to type bool which takes a value true or false.
- Examples:
  - X=10; y=15; X <= Y
  - A=2; b=3; c=2; d=4; A+b == c+d
  - 10<25
  - M=10; n=20; M+n>100
  - 10.5<10

Logical expressions

- Logical expressions combine two or more relational expressions and produces bool type results.
- Example:
  - A>b && x==10
  - X==10 || y==5

Bitwise expressions

- Bitwise expressions are used to manipulate data at bit level.
- Basically used for testing or shifting bits.
- Examples:
  - X<<3; //shift three bit position to left.
  - Y>>1; //shift one bit position to right.

Special Assignment Expressions

- Chained Assignment
- Embedded Assignment
- Compound Assignment

Chained Assignment

- Assigning same to more than one variable is called chained assignment.
- Example:
  - X= (y=10);
  - Or
  - X=y=10;

  - Float a = b= 12.4; //wrong

  - Float a, b;
  - a = b = 12.34; //correct

  - Float a=12.34, b=12.34; //correct

- A chained statement cannot be used to initialize variables at the time of declaration.

Embedded Assignment

- X=(y=50)+10; // x=60

- Y=50;
- X=y+10; //x=60

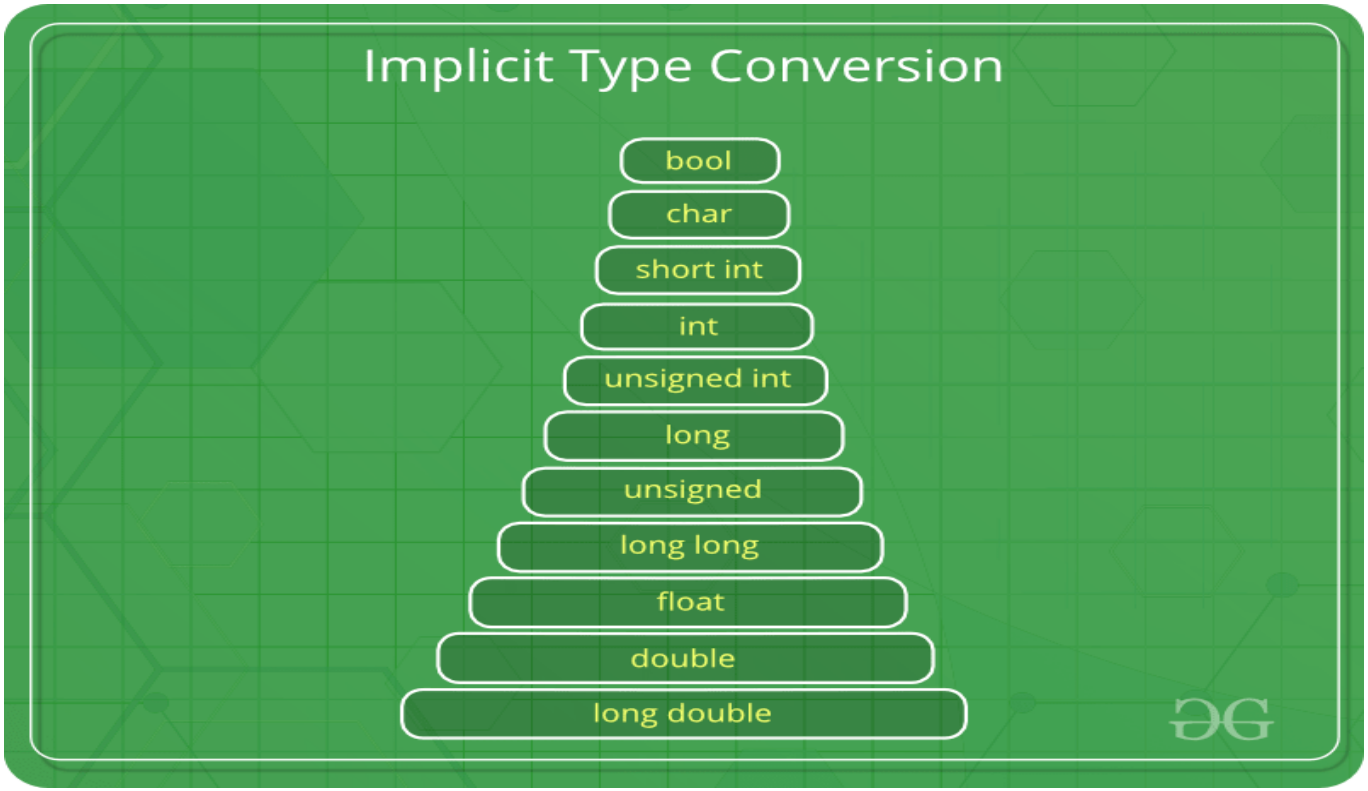
Compound Assignment

- Compound assignment operator → +=
- Example:
  - X += 10; // x = x+10
- Syntax:
  - Variable1 op= variable2;
  - Variable1 = variable1 op variable2;

- Examples:
  - X += y; // x=x+y
  - X -= y; //x=x-y
  - X *= y; //x=x*y
  - X /= y; //x=x/y

Implicit Conversion

- Automatic or implicit conversion
- Water fall model: smaller type to wider type



Mixed – mode operations

- Int x = int y + float z;
- int x = 10 + 12.5; // x=22 (x integer – explicit)

- X = 10+12.5; //x=22.5 (x float – implicit)

- Integral widening conversion

Operator overloading

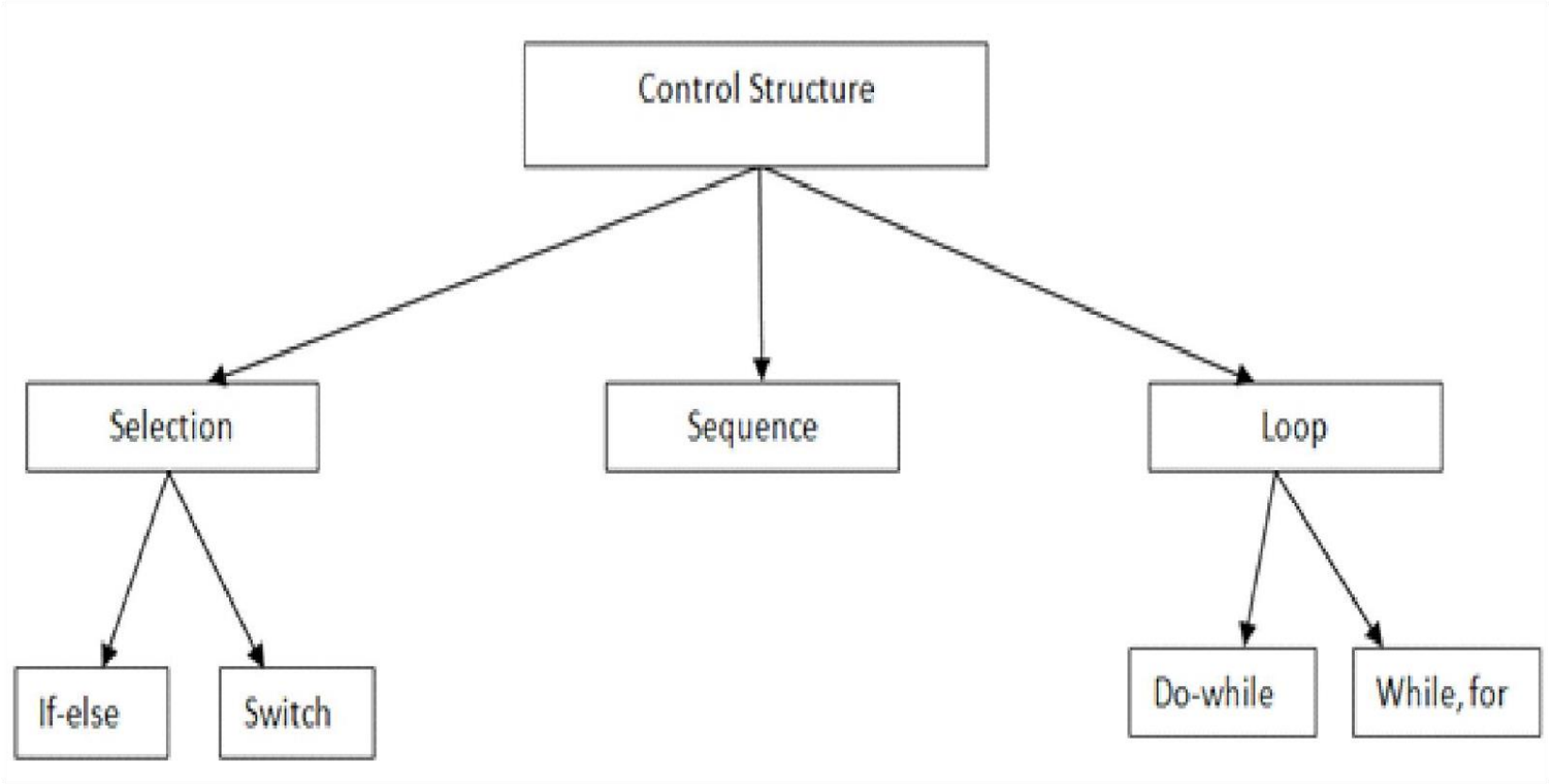
- Assign multiple meaning to operators.
- <<
- Example:
  - cout<<75.86; //integer op
  - Cout<<“Hello”; //string op
- Member access operators: (. and *)
- Conditional operator: (?:)
- Scope resolution operator: (::)
- Size operator: (sizeof())

Operator Precedence

- BODMAS
  - Brackets
  - Of
  - Division
  - Multiplication
  - Addition
  - Subtraction

Precedence	Operator	Description	Associativity
1	::	Scope resolution	Left-to-right
2	++ -- () [] . ->	Suffix/postfix increment and decrement Function call Array subscripting Element selection by reference Element selection through pointer	
3	++ -- + - ! ~ (*type*) * & sizeof new, new[] delete, delete[]	Prefix increment and decrement Unary plus and minus Logical NOT and bitwise NOT Type cast Indirection (dereference) Address-of Size-of Dynamic memory allocation Dynamic memory deallocation	Right-to-left
4	.* ->*	Pointer to member	Left-to-right
5	* / %	Multiplication, division, and remainder	
6	+ -	Addition and subtraction	
7	<< >>	Bitwise left shift and right shift	
8	< <= > >=	For relational operators < and ≤ respectively For relational operators > and ≥ respectively	
9	== !=	For relational = and ≠ respectively	
10	&	Bitwise AND	
11	^	Bitwise XOR (exclusive or)	
12	|	Bitwise OR (inclusive or)	
13	&&	Logical AND	
14	||	Logical OR	
15	?: = += -= *= /= %= <<= >>= &= ^= |=	Ternary conditional Direct assignment (provided by default for C++ classes) Assignment by sum and difference Assignment by product, quotient, and remainder Assignment by bitwise left shift and right shift Assignment by bitwise AND, XOR, and OR	Right-to-left
16	throw	Throw operator (for exceptions)	
17	,	Comma	Left-to-right

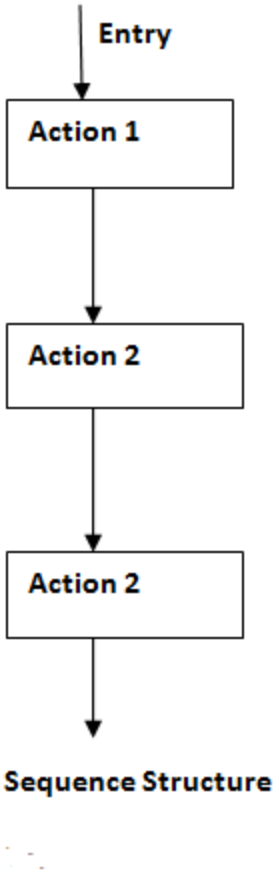
Basic Control Structures



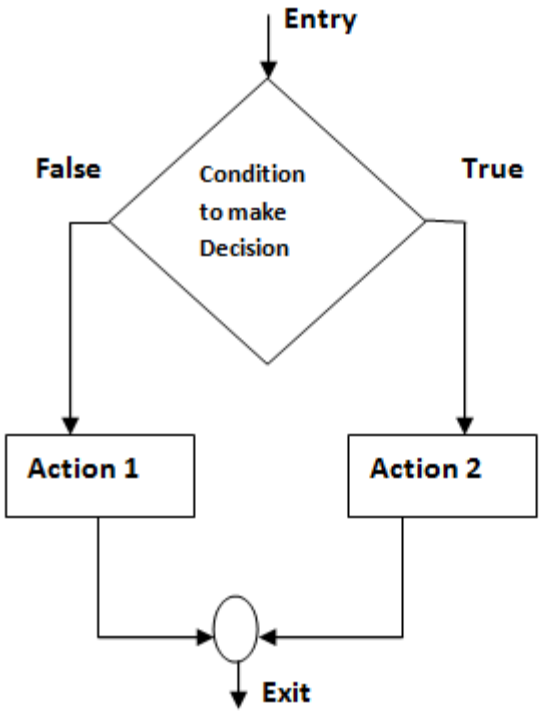
Basic Control Structure

- Sequence Structure (straight line)
- Selection Structure(Branching)
- Loop Structure (iteration or repetition)

Sequence Structure



Selection Structure

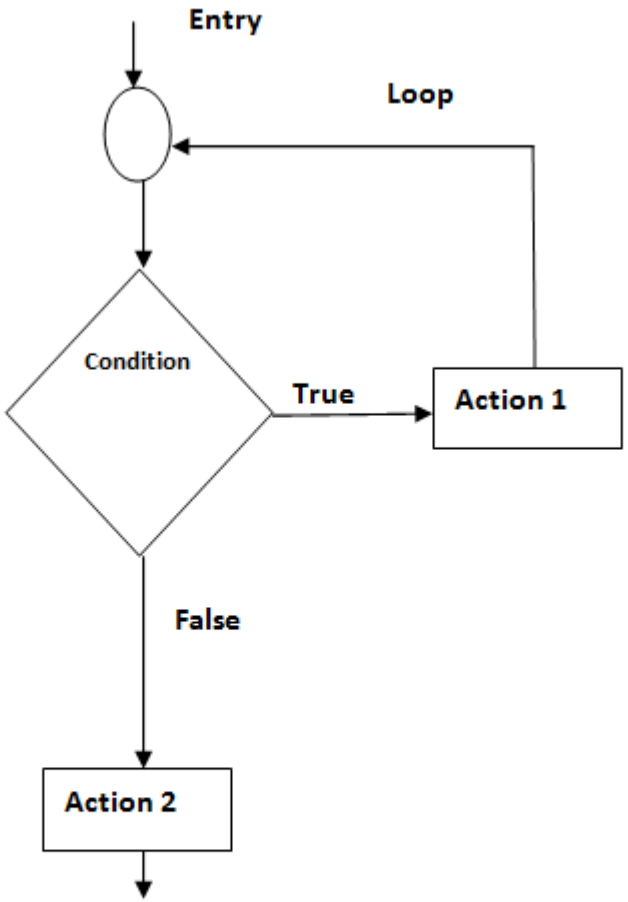


Selection Structure

Implemented using:- If and If..else control statements

switch is used for multi branching

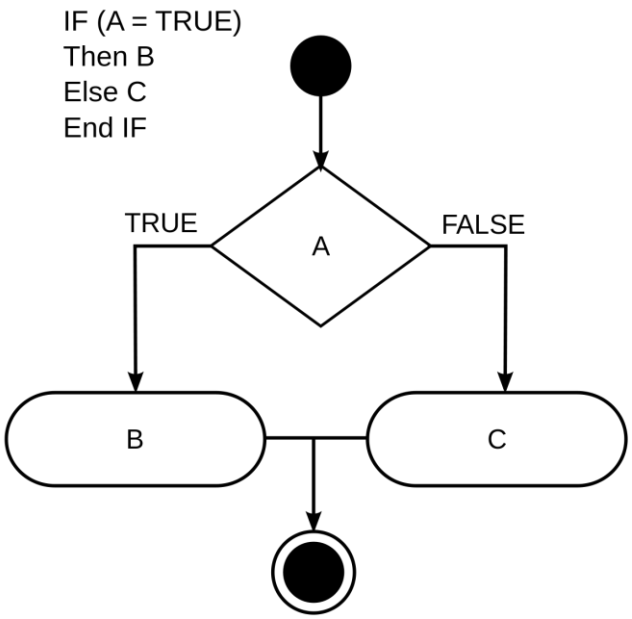
Loop Structure

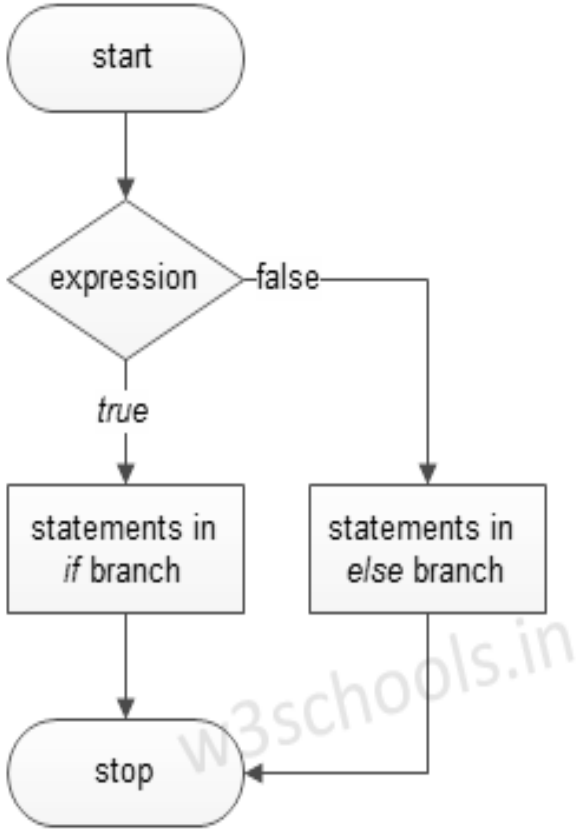


Loop Structure

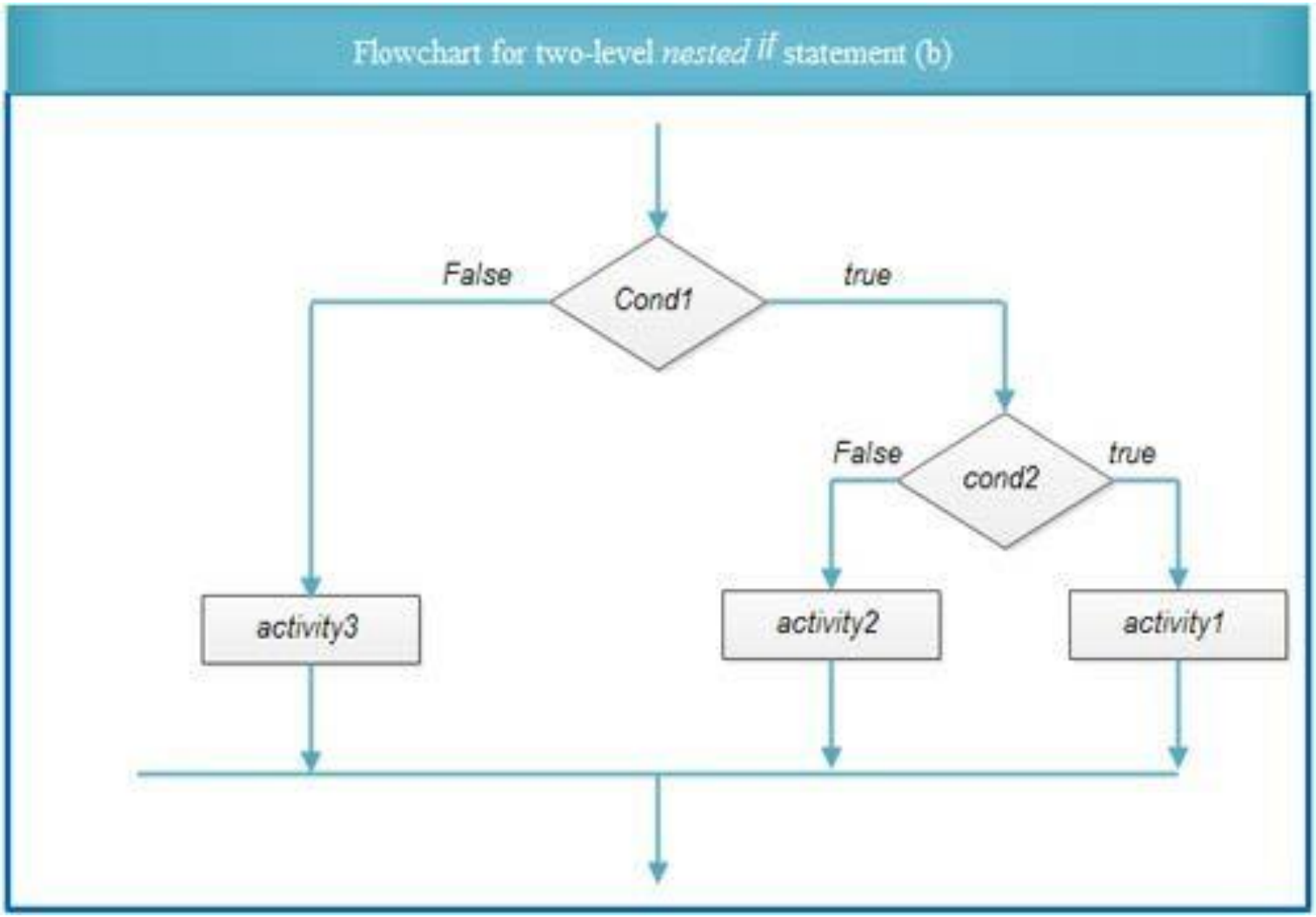
Implemented using:- While , Do While and For control statements

First level abstraction

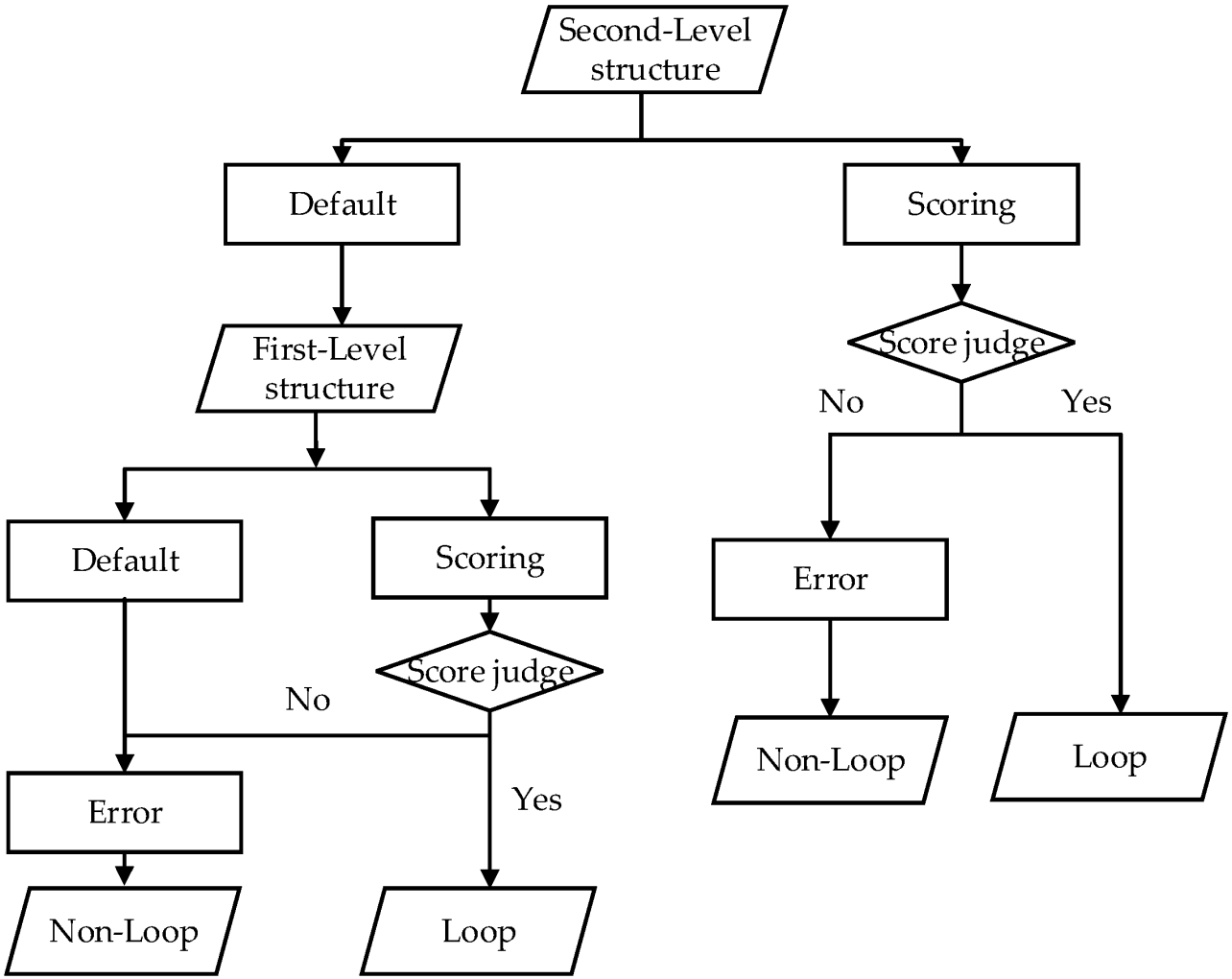




Second level of Abstraction



Detailed level of abstraction



The if statement

- Simple if statement
- Form1:

```
if (expression is true)
{
    action1;
}
Action2;
Action3;
```

- If..else statement
- Form2:

```
if (expression is true)
{
    action1;
}
else
{
    action2;
}
Action3;
```

Switch statement

- Multiple branching statement, where based on a condition, the control is transferred to one of the many possible blocks.
- Syntax:

```
switch(expression)
{
    case1:
    {
        action1;
    }
    case2:
    {
        action2;

    }
    case3:
    {
        action3;
    }
    default:
    {
        defaultaction4;
    }
}
action5;
```

Do-while statement

- Exit-controlled loop.
- Based on a condition, the control is transferred to a particular point of a program.
- Syntax:

```
do
{
   action1;
}while (condition is true);
action2;
```

While statement

- Entry-controlled loop
- Syntax:

```
While (condition is true)
{
   action1;
}
action2;
```

The for statement

- Entry-controlled loop.
- It is used when an action is to be repeated for a predetermined number of times.
- Syntax:

```
for (initial value; test; increment)
{
   action1;
}
action2;
```

Chapter 4

Functions in C++

Introduction

- Dividing a program into functions is one of the major principles of top-down, structured programing.
- Example:

```
void show ( ) ;
main ( )
 {
        . . . . .
        show ( ) ;
        . . . . .
 }
  void show ( )
 {
        . . . . .
        . . . . .
        . . . . .
 }
```

- When the function called, control is transferred to the first statement in the function body. The other statement in the function body are then execute and control returns to the main program when the closing brace is encountered.

The Main Function

- C does not specify any return type for the main () function which is the starting point for the executing of a program.
- Example:

```
main ( )
{
        // main programing statements
 }
```

- Function should return a value.

Function Prototyping

- The prototype describe the function interface to the compiler by giving details such as the number and the type of arguments and the type of return values. With function prototyping, a template is always used when declaring and defining a function. When a function is called, the compiler uses the template to ensure that proper arguments are passed, and the return value is treated correctly.
- Syntax:

  type function-name (argument-list) ;
- Example:

  float volume (int x, float y, folat z) ;

Call By Reference

- A function call passes arguments by value.
- The ‘called function’ creates a new set of variables and copies the values of arguments into them. The function does to have access to the actual variables in the calling program and can only work on the copies of values. This mechanism is fine if the function does not need to alter the values to the original variables in the calling program.
- But, there may arise situations where we would like to change the values of variables in the calling program.
- When we pass arguments by reference, the ‘formal’ arguments in the called function become aliases to the ‘actual’ arguments in the calling function.
- Example:
  - Void swap(int &a, int &b)
  - {

```
int t=a;
a=b;
b=t;
}
```

Return by Reference

- A function can also return a reference.
- Example:

```
int &max( int &x, int &y)
{
    if (x>y)
            return x;
    else
            return y;
}
```

Inline Functions

- A function that is expanded in line when it is invoked.
- Syntax:

```
inline function-header
{
    Function body
}
```

- Example:

```
Inline double cube ( double a)
{
    return (a*a*a);
}
```

Situations where inline may not work

1. For functions returning values, if a loop, a **switch**, or a **goto** exists.
2. For functions not returning values, if a return statement exists.
3. If functions contain **static** variables.
4. If **inline** functions are recursive.

Default Arguments

- Default values are specified when the function is declared.
- Example:
  - float amount (float principal, int period, float rate=0.15);
  - Value = amount(5000,7);
  - Value = amount(5000, 5, 0.12);
- Advantages:
  - We can use default arguments to add new parameters to the existing functions.
  - Default arguments can be used to combine similar functions into one.

const Arguments

- An argument to a function can be declared.
- Examples:
  - int strlen(const char *p);
  - int length(const string &S);

Recursion

- Recursion is a situation where a function calls itself i.e., on e of the statements in the function definition makes a call to the same function in which it is present.
- It may sound like an infinite looping condition but just as a loop has a conditional check to take the program control out of the loop, a recursive function also possesses a base case which returns the program control from the current instance of the function to call back to the calling function.

Recursion - example

//Calculating factorial of a Number.

```
#include<iostream>
#include<conio.h>
using namespace std
 long fact(int n)
{
        if (n==0)
                return 1;
        else
                return (n*fact(n-1));
}
int main()
{
        int num;
        cout<<“Enter a positive integer:”; cin>>num;
        cout<<“Factorial of “<<num<<“is:”<<fact(num);
        getch();
        return 0;
}
```

Model Output

Enter a positive integer: 10

Factorial of 10 is : 362880

Function Overloading

- Overloading refers to the use of the same thing for different purposes.
- C++ also permits overloading of functions.
- This means that we can use the same function name to create functions that perform a variety of different tasks. This is known as *function polymorphism* in OOP.

Function overloading - example

//declaration

- int add(int a, int b);
- int add (int a, int b, int c);
- double add (double x, double y);
- double add (int p, double q);
- double add (double p, int q);

//function calls

- cout<<add(5, 10);
- cout<<add(5, 10, 15);
- cout<<add(12.5, 10.5);
- cout<<add(5, 10.5);
- cout<<add(12.5, 10);

Function Selection - Steps

1. The compiler first tries to find an exact match in which the types of actual arguments are the same, and use that function.
2. If an exact match is not found, the compiler uses the integral promotions to the actual arguments, such as char to int or float to double to find a match.
3. When either of them fails, the compiler tries to use the built-in conversions (the impilict assignment conversions) to the actual arguments and then uses the function whose match is unique. If the conversion is possible to have multiple matches, then the compiler will generate an error message.
4. If all the stepts fail, then the compiler will try the user-defined conversions in combination with integral promotions and built-in conversions to find a unique match. User-defined conversions are often used in handling class objects.

Friend and Virtual Function

- Two new functions introduced in C++:
  - Friend function
  - Virtual function
- They are basically introduced to handle some specific tasks related to class objects.

Math Library Function

- Math functions that can be used for performing certain commonly used calculations.

*In the following list, x and y are of type double, n is an int, and all functions return double. All angles MUST be in radians.**

sin(x)	// sine of x
cos(x)	// cosine of x
tan(x)	// tangent of x
asin(x)	// $\sin^{-1}(x)$ in range $[-\pi/2, \pi/2]$, $x \in [-1, 1]$
acos(x)	// $\cos^{-1}(x)$ in range $[0, \pi]$, $x \in [-1, 1]$
atan(x)	// $\tan^{-1}(x)$ in range $[-\pi/2, \pi/2]$
atan2(y,x)	// $\tan^{-1}(y/x)$ in range $[-\pi, \pi]$
sinh(x)	// hyperbolic sine of x
cosh(x)	// hyperbolic cosine of x
tanh(x)	// hyperbolic tangent of x
exp(x)	// exponential function e^x
log(x)	// natural logarithm ln()x), x>0
log10(x)	// base 10 logarithm $\log_{10}(x)$, x>0
pow(x,y)	// x^y. A domain error occurs if x=0 and y<=0, or if x<0 and y is not an integer.
sqrt(x)	// $\sqrt{x}$ x >= 0
ceil(x)	// smallest integer not less than x, as a double
floor(x)	// largest integer not greater than x, as a double
fabs(x)	// absolute value |x|

These two are found in <stdlib.h> or <cstdlib>

abs(iVal)	// absolute value |iVal|. iVal is int.
labs(lVal)	// absolute value |lVal|. lVal is long.

Chapter 5

Classes and Objects

Introduction

- A Class is an extension of the idea of structure used in c. it is a new way of creating and implementing a user-defined data type.

C Structures Revisited

- Structures provide a method for packing together data of different types.
- A structure is a convenient tool for handling a group of logically related data items.
- It is a user-defined data type with a template that serves to define its data properties.
- Once the structure type has been defined, we can create variables of that type using declarations that are similar to the built-in type declarations.

```
struct student
{
        char name[20];
        int roll-number;
        float total-marks;
}

struct student A;

strcpy(A.name=“John”);
A.roll-number=123;
A.total-makrs=400;
```

Limitations of C Structure

- Cannot add two Complex numbers.
- Do not permit data hiding.
- Ex.

```
struct complex
{
        float x;
        float y;
}
struct complex c1, c2, c3;
C3 = c1 + c2;
```

Extensions to C Structures

- Inheritance: inherits characteristics from other types.
- A structure can have both variables and functions as members.
- The keyword *struct* can be omitted in the declaration of structure variables.

Specifying a Class

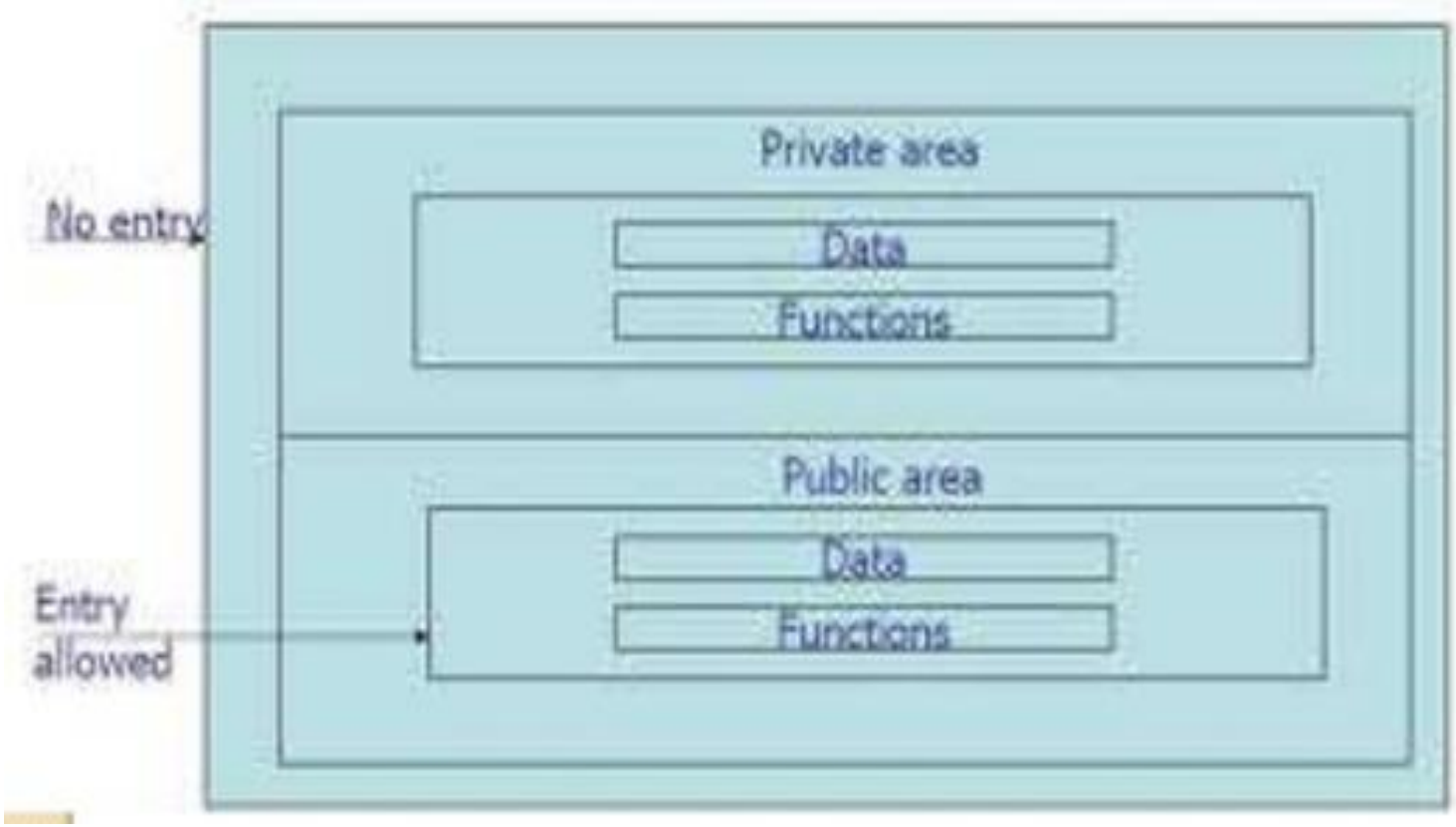
A class specification has two parts:

1. Class declaration
2. Class function definitions.

Syntax:

```
class class-name
{
        private:
                variable declarations;
                function declarations;
        public:
                variable declarations;
                function declarations;
};
```

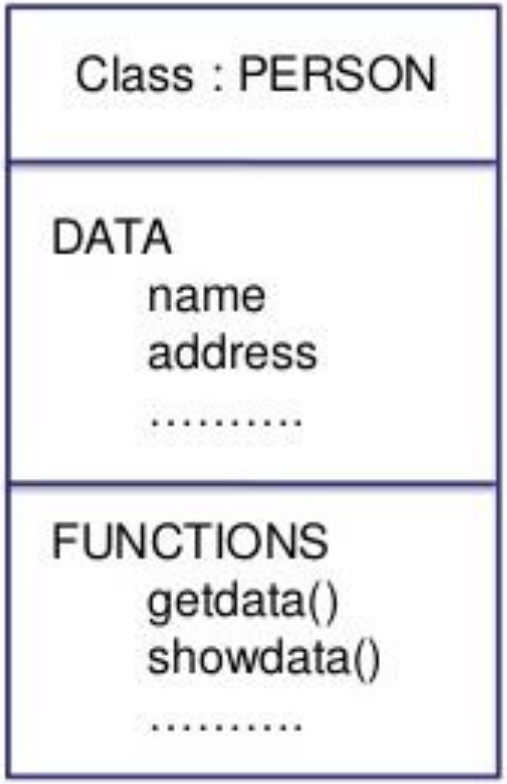
Data hiding in Classes

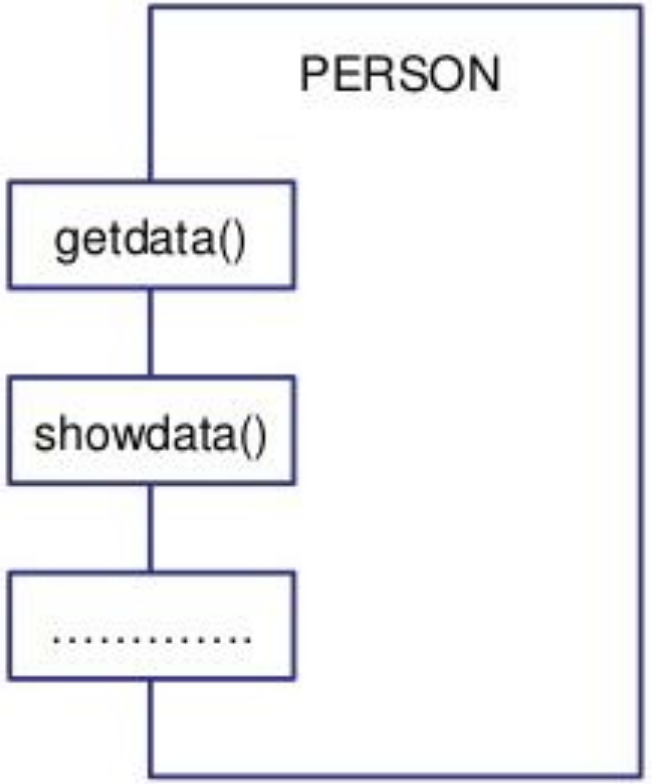


A Simple Class Example

```
class item
{
        int number;
        float cost;
        public:
                void getdata(int a, float b);
                void putdata(void);
};
```

Creating Objects





Accessing Class Members

- The private data of a class can be accessed only through the member functions of the class.
- Syntax:

  object-name . Function-name (actual-arguments);

- Example:

  x.getdata(100, 75.5);

Defining Member functions

- Member functions can be defined in two places:
  - Outside the class definition
  - Inside the class definition

Outside the Class Definition

```
return types class-name :: function-name
(argument declaration)
{
        function body;
}

void item :: getdata (int a, foat b)
{
        number =a ;
        cost =b;
}
void item :: putdata(void)
{
        cout<<“Number”<<number;
        cout<<“Cost”<<cost;
}
```

- An important difference between a member function and a normal function is that a member function incorporates a membership ‘identity label’
- Several different classes can use the same function name. the ‘membership label’ will resolve their scope.
- Member functions can access the private data of the class. A nonmember functin cannot do so. (However, an exception to this rule is a friend function.
- A member function can call another member function directly, without using the dot operator.

Inside the Class Definition

- Another method of defining a member function is to replace the function declaration by the actual function definition inside the class.
- When a function is defined inside a class. It is treated as an inline function. Therefore, all the restrictions and limitations that apply to an inline function are also applicable here.

```
class item
{
        int number;
        float cost;
        public:
                void getdata (int a,float b);
        void putdata (void)
        {
                cout<<number;
                cout<<cost;
        }
};
```

C++ Program with Class

```
#include<iostream>
Using namespace std;
class item
{
        int number;
        float cost;
        public:
                void getdata (int a,float b);
        void putdata (void)
        {
                cout<<number;
                cout<<cost;
        }
};
void item :: getdata(int a, float b)
{
        number =a ;
        cost = b;
}
```

```
int main()
{
        item x;
        cout<<“object x”;
        x.getdata(100,  299.95);
        x.putdata();
        item y;
        cout<<“object y”;
        y. getadata(200, 175.50);
        y.putdata();
        return 0;
}
```

Model Output:

Object x

Number:100

Cost: 299.95

Object y

Number : 200

Cost : 175.5

Making an Outside Function inline

We can define a member function outside the class definition and still make it inline by just using the qualifier inline in the header line of function definition.

```
class item
{
        .....
        .....
        public:
                void getdata(int a, float b);
};
Inline void item::getdata(int a, float b)
{
        number=a;
        cost =b;
}
```

Nesting of Member Functions

A member function can be called by using its name inside another member function of the same class. This is known as nesting as member function.

```
#include<iostream>
#include<conio.h>
#include<string>
using namespace std;
class binary
{
        string s;
        public:
        void read(void)
        {
                cout<<“Enter a binary
number”;
        cin>>s;
}
Void chk-bin(void)
{
        for (int i=0; i<s.length(); i++)
        {
                if(s.at(i)!=‘0’ && s.at(I)!=‘1’)
```

Private Member Functions

A private member function can only be called by another function that is a member of its class. Even an object cannot invoke a private function using the dot operator.

```
class sample
{
        int m;
        void read(void);
        public:
                void update(void);
                void write(void);
};

S1.read();//won't work. Objects cannot access private members.

void sample :: update(void)
{
        read();                 //simple call.no objects used.
}
```

Arrays within a Class

The arrays can be used as member variables in a class.

```
const  int size=10;
class array
{
        int a[size];
        public:
                void setval(void);
                void display(void);
};
```

Memory Allocation for Objects

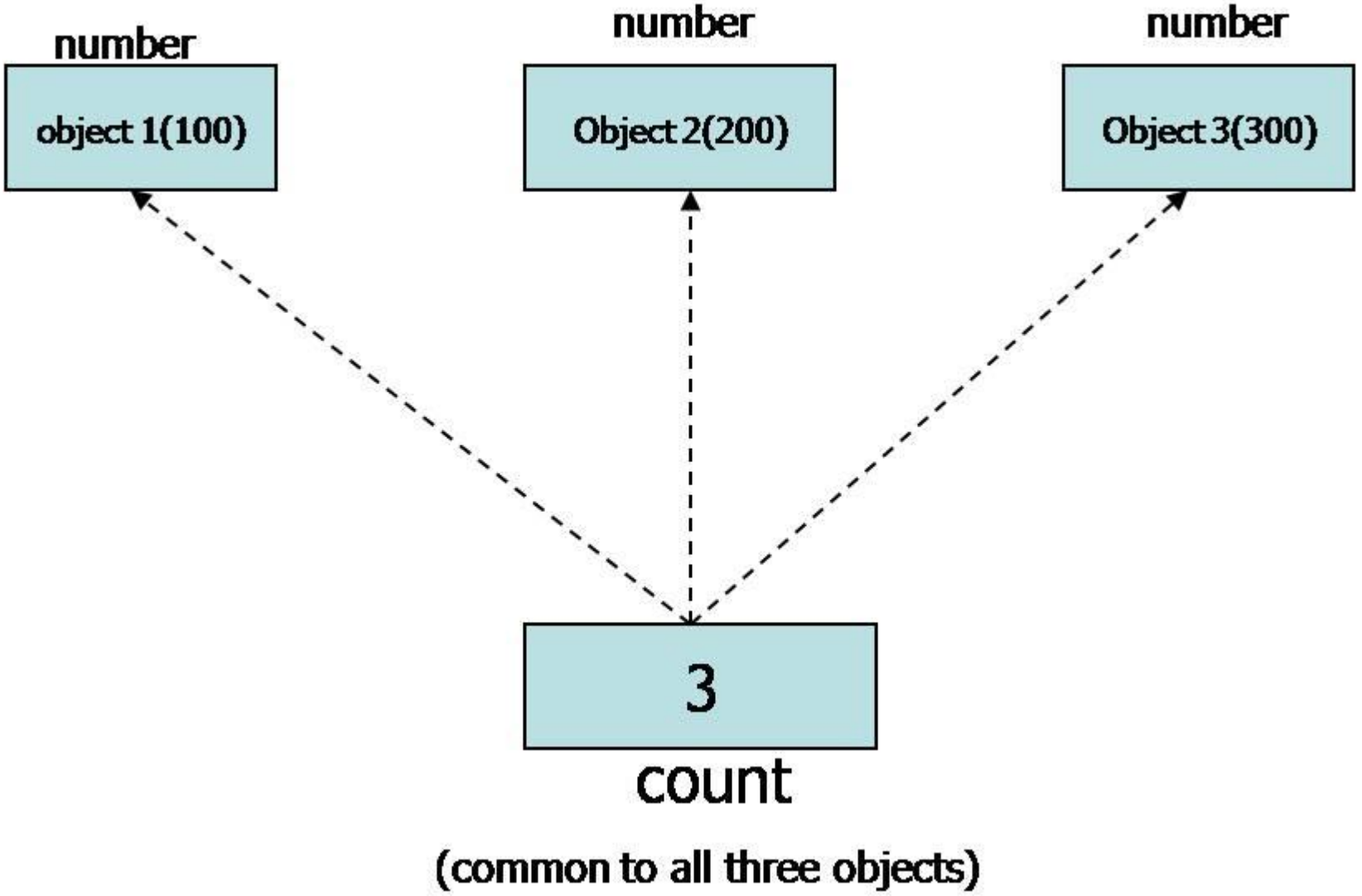
The member functions are created and placed in the memory space only once when they are defined as a part of a class specification.

Since all the objects belonging to that class use the same member functions, no separate space is allocated for member functions when the objects are created.

Static Data Members

- A data member of a class can be qualified as static. The properties of a static member variable are similar to that of a C static variable. A static member variable has certain special characteristics.
- Example:
  - static int count;
- It is iitialized to zero when the first object of its class is created. No other initialization is permitted.
- Only one copy of that member is created for the entire class and is shared by all the objects of that class, no matter how many objects are created.
- It is visible only within the class, but its lifetime is the entire program.

Sharing of a static data member



Static Member Functions

- A static function can have access to only other static members (functions or variables) declared in the same class.
- A static member function can be called using the class name (instead of its objects)
- Syntax:

  class-name :: function-name
- Example:

```
static int count;
```

Arrays of Objects

- Arrays of variables that are of the type class.
- Example:

```
class employee
{
        char name[30];
        float age;
        public:
        void getdata(void);
        void putdata(void);
}
employee manager[13];
employee foreman[15];
employee worker[75];
```

Objects as Function Arguments

- Like other data type, an object may be used as a function argument.
- 2 methods:
- Pass-by-value
  - A copy of the entire object is passed to the function.
- Pass-by-reference
  - Only the address of the object is transferred to the function.

Example:

```
class time
{
        public:
        void sum(time, time);
}
int main()
{
        time t1,t2, t3;
        t3.sum(t1,t2);
}
```

Friendly Functions

- To make an outside function “friendly” to a class, we have to simply declare this function as a friend.
- Characteristics:
  - It is not in the scope of the class to which it has been declared as friend.
  - Since it is not in the scope of the class, it cannot be called using the object of that class.
  - It can be invoked like a normal function without the help of any object.
  - Unlike member functions, it cannot access the member names directly and has to use an object name and do membership operator with each member name.
  - It can be declared either in the public or the private part of a class without affecting the meaning.
  - Usually, it has the objects as arguments.

Friendly functions

Syntax:

```
class abc
{
        public:

        friend void xyz(void);
}
```

Example:

```
class sample
{
public:
friend float mean(sample s);
};
float mean (sample a)
{
return float (s.a+s.b)/2.0;
}
```

Returning Objects

- A function cannot only receive objects as arguments but also can return them.

Example:

```
class sample
{
        public:
        int sum(int a, int b);
};
int sum(int a, int b)
{
        return a+b;
}
```

Chapter 6

Constructors and Destructors

Introduction

- Constructors enables an object to initialize itself when it is created. This is known as automatic initialization of objects.
- It also provides another member function called the destructor that destroys the objects when they are no longer required.

Constructors

- A constructor is a ‘special’ member function whose task is to initialize the objects of its class.
- Constructor’s name is the same as the class name.
- The construtor is invoked an object of its associated class is created.
- It is called constructor because it constructs the values of data members of the class

```
class integer
{
        int m, n;
        public:
                integer(void);
};
integer::integer(void)
{
        m=0; n=0;
}
```

- Characteristics:
- They should be declared in the public section.
- They are invoked automatically when the objects are created.
- They do not have return types, not even void and therefore, and they cannot return values.
- They cannot be inherited, though a derived class can call the base class constructor.
- Like other C++ functions, they can have default arguments.
- Constructors cannot be **virtual**.
- We cannot refer to their addresses.
- An object with a constructor (or destructor) cannot be used as a member of a union.
- They make ‘implict calls’ to the operators **new** and **delete** when memory allocation is required.

Parameterized Constructors

- The constructors that can take arguments are called parameterized constructors.
- 2 ways:
  - By calling the constructor explicitly.
  - By calling the constructor implicitly.
- Example:
  - integer int1(100, 150);

Multiple Constructors in a Class

- A class can have multiple constructors that assign the fields in different ways. Sometimes it's beneficial to specify every aspect of an object's data by assigning parameters to the fields, but other times it might be appropriate to define only one or a few.
- One class may have more than one constructors.
- Multiple constructors are used to initialize different sets of class attributes.
- When a class has more than one constructors. It is call constructor overloading.
- Constructors those receive parameters are called parameterized constructors.

```
#include <iostream>
using namespace std;
class ABC
{
  private:
      int x,y;
  public:
       ABC ()                    //constructor 1 with no arguments
 {
        x = y = 0;
  }
ABC(int a) //constructor 2 with one argument
    {
        x = y = a;
     }
     ABC(int a,int b) //constructor 3 with two argument
     {
          x = a; y = b;
      }
      void display()
      {
           cout << "x = " << x << " and " << "y = " << y << endl;
      }
 };


int main()
 {
      ABC cc1; //constructor 1
      ABC cc2(10); //constructor 2
      ABC cc3(10,20); //constructor 3
      cc1.display();
      cc2.display();
      cc3.display();
      return 0;
 } //end of program

Model Output:
x = 0 and y = 0
  x = 10 and y = 10
  x = 10 and y = 20
```

Constructors with Default Arguments

- Default argument constructor can be called with either one argument or no arguments.
- When called with no arguments, it becomes a default constructor.

- Limitations:
  1. Constructors do not have any return type.
  2. Constructors can neither be used as virtual nor inherited.
  3. Constructors should be declared in public section only.
  4. Constructors' memory address cannot be fetched.

Dynamic Initialization of Objects

The dynamic initialization of object means to the data members of the class while creating the object when we want to provide initial or default values to the data members while creating of object we need to use dynamic initialization of object.

Model Output:

Enter roll number to initialize the object:101
Enter percentage to initialize the object :84.02
After initializing the object the values are…
Roll number:101
percentage:84.02%
Enter roll number:102
Enter percentage:87

Enter number:102
Percentage :87%

```
#include<iostream>
      using namespace std;
      struct student
       {
         private:
            int r No;
            float price;
         public:
            student(int r ,float p)
             {
                        r. No =r;
                        price =p;
             }
    void read (void)
    {
              cout <<"Enter roll number:";
              cin >>r No;
              cout<<"Enter percentage:";
              cin>> percen;
     }
     void print(void)
      {
              cout<<end1;
              cout<<"Roll number:"<< r No<<end1;
              cout<<"percentage:"per<<"%"<<end1;
    }
 };

int main()
 {
  cout<<"Enter roll number to initialize the
                                        object:";
std . read();
  std . print();
   return 0;
}
```

Copy Constructors

- A copy constructor is used to declare and initialize an object from another object.
- The process of initializing through a copy constructor is known as copy initialization.
- A copy constructor takes a reference to an object of the same class as itself as an argument.
- Example:
  - integer(integer &i);

Dynamic Constructors

- The constructors can also be used to allocate memory while creating objects . This will enable the system to allocate the right amount of memory for each object when the objects are not of the same size, thus resulting in the saving of memory . Allocation of memory to objects at the run time of their construction is known as dynamic construction of objects . The memory is allocated with the help of the' new 'operator.
- In constructors that are used to construct strings in objects

Model Output:

Joseph
Louis
Lagrange
Joseph Louis
Joseph Louis Lagrange

```
#include<iostream>
  #include<string>
   using namespace std;
    class String
    {
         char  *name;
          int  length;
       public:
              String()                //constructor-1
{
              length = 0;
              name = new char[length +1];
     }
      String(char *s)    //constructor-2
      {
           length =  strlen(s);
           name = new char[length +1];
           strcpy(name, s);
       }
        void display (void)
        { cout  << name <<  "\n" ;}
void join(String &a, String &b);
 };
 void String :: join(String &a, String  &b)
 {
      length = a . length + b . length;
      delete name;
      name = new char[length+1];   //dynamic allocation

       strcpy(name, a. name);
       strcat(name, b. name);
};
int main()
   {
        char  *first  =  "  Joseph ";
        string name1(first),name2(" Louis "),name3(
        "Lagrange "),s1,s2;
         s1.join(name1,name2);
         s2.join(s1,name3);
         name1.display();
         name2.display();
         name3.display();
s1.display();
        s2.display();
        return 0;
 }
```

Constructing Two-Dimensional Arrays

- We can construct matrix variables using the class type objects.

const Objects

- A const member is a function prototype or function definition where the keyword const appears after the function's signature.

Destructors

- A destructor, as the name implies, is used to destroy the objects that have been created by a constructor.

- Syntax:
  - ~classname() {}
- Example:
  - ~integer() {}
  - matrix :: ~matrix()

```
{
    delete p;
}
```

Chapter 7

Operator Overloading and Type Conversions

Introduction

- C++ has the ability to provide the operators with a special meaning for a data type.
- The mechanism of giving such special meanings to an operator is known as operator overloading.

- Class member access operators(., .*)
- Scope resolution operator (::)
- Size operator (sizeof)
- Conditional operator (?:)

Defining Operator Overloading

- Syntax:

```
return-type classname ::
operator op(arglist)
{
        function body;
}
```

- Process steps:

1. Create a class that defines the data type that is to be used in the overloading operation.
2. Declare the operator function operator op() in the public part of the class. It may be either a member function or a friend function.
3. Define the operator function to implement the required operations.

Some other Operator Overloading Example

- Overloading the Subscript Operator []
  - Used to access and modify a specific element in an array.
- Overloading the Pointer-to-member (->) Operator
  - Used in conjunction with an object pointer to access any of the object's members.

Rules for Operator Overloading

1. Only existing operators can be overloaded. New operators cannot be created.
2. The overloaded operator must have at least one operand that is user-defined type.
3. Overloaded operators follow the syntax rules of the original operators. They cannot be overridden.
4. There are some operators that cannot be overloaded.
5. We cannot use friend functions to overload certain operators. However, member functions can be used to overload them.
6. Unary operators, overloaded by means of a member function, take no explicit arguments and return no explicit values, but, those overloaded by means of a friend function, take one reference argument (the object of the relevant class).
7. Binary operators overloaded through a member function take one explicit argument and those which are overloaded through a friend function take two explicit arguments.
8. When using binary operators overloaded through a member function, the left-hand operand must be an object of the relevant class.
9. Binary arithmetic operators such as +, -, *, and / must explicitly return a value. They must not attempt to change their own arguments.

Rules for Operator Overloading

- C++ operators that can be overloaded

Operators that can be overloaded							
+	-	*	/	%	^	&	|
~	!	=	<	>	+=	-=	*=
/=	%=	^=	&=	|=	<<	>>	>>=
<<=	==	!=	<=	>=	&&	||	++
--	->*	,	->	[]	()	new	delete
new[]	delete[]						

- C++ Operators that cannot be overloaded

Operators that cannot be overloaded				
.	.*	::	?:	sizeof

Limitations:

1. Some of the operators like ::, ->, and Sizeof() cannot be overloaded.
2. By operator overloading, you cannot change the precedence, associative and number of arguments of an operator.