C++

by Examples

A Complete Material on C++ Language Programming.

by Srihari Bezawada

Since 1999

C-Family Computer Education

Flyover Pillar No:16, Service Road, Benz Circle,
Vijayawada, AP, India, pincode-520010,
94400-30405, 8500-117118.

This book is designed for those who are proficient in C and want to learn C++. It focuses on explaining the internal workings of object-oriented programming through well-known examples. The examples provided are relevant to everyday life, presented in a straightforward way, and same examples are repeated in all chapters to help readers focus on concepts rather than coding.

C++ with OOP

A complete reference material on C++ with object oriented programming

About C

C-language is an ideal programming language and a perfect combination of power and flexibility. Though it is a high-level language, it has low level features as well as easy interaction with hardware and OS. Even assembly language code can be mixed in it. It is accepted as a good system programming language. Powerful OS like UNIX was developed using C.

'C' is widely used in the software industry for both system and application side development. C is the primary subject in computer science. It lays good foundation in programming and eases the learning of any advanced courses that are to follow. So, it has been added in every branch of arts, science, and engineering as part of curriculum. Every graduate student is going to face C, C++, DS, Java, Python and Oracle in the first & second academic years.

Now a day, some students and colleges prefer Python or Java as a beginner's language, but based on my 30 years of experience, I always recommend starting with C, Data Structures (DS), and C++. These provide the best foundation for beginners. Once students master these, updating to Java or Python becomes much easier and more time-efficient.

C++ lays a strong foundation in object-oriented programming. Learning the basics of C++ typically takes no more than 15 hours. I'm always surprised that, these days, parents—despite having no programming knowledge—often decide what their children should learn.

About C++

C++ is a successor to the C language. It introduces you to the exciting world of object-oriented programming. Once you experience the magic of object-oriented programming, you'll never want to leave it. Among all the available object-oriented programming languages, C++ stands out as the ideal choice. With C++, you can create portable class components that can even be reused in other languages. Thanks to these advantages, C++ is widely used in multi-tier enterprise applications.

About C-Family Computers

C-Family is a premier institute in Vijayawada training the students in C, C++, DS, Java, Python, Oracle, Web Design courses **since 1999**. C-family was not setup by any business men instead it was setup and run by a group of eminent programmers. We provide hands on training to ambitious students in right proportions to their passion for knowledge. Needless to say, taking individual care of each student is the hallmark of our institution. Our courses are designed in such a way that student get best foundation in logic and programming skills. [The languages C, C++, VC, VC++, JAVA are called C-Family Languages, as we teach all these courses, we named this institute **C-Family Computer Education**]

Introduction to C++

C is both a structured and function-oriented language. Structured means that the language provides good control statements along with building blocks of code defined by braces **{ }.** The control statements such as if, if-else, while, for, switch, break, continue, return, etc are provided with a great flexible syntax. Using these statements, instructions are built in a structured way so that one can easily understand the control flow.

Before the C language, older languages like BASIC, Fortran, Algol, and Pascal lacked proper control statements and were considered to be unstructured. The code in these unstructured languages became difficult to read and manage as it grew larger. As a next-generation language, C introduced robust control statements and building blocks defined by braces **{}**. These are made C a well-structured language.

In C, we break down a large program into several subprograms called functions. This allows us to express a large program as a collection of functions. Thus, the entire programs are made up of a collection of functions. Therefore, C is often referred to as a function-oriented or procedure-oriented language. In Pascal or older languages functions are called procedures. Functions are also called sub-programs, routines, subroutines, methods, or modules. Finally, C is said to be both a structured and function-oriented language.

The main issue with function-oriented languages is that all functions are defined globally, meaning any function can be called from any part of the program. As the number of functions increases in larger projects, managing them becomes increasingly complex. Once the number of functions exceeds 100, it becomes difficult to remember and organize them effectively. In fact, the C language tends to struggle when the program size exceeds 50,000 lines of code.

The C language is well-suited for solving algorithmic problems using functions. However, as the amount of data and the number of functions increase, it becomes difficult to manage them. Therefore, C is more suitable for system and scientific applications rather than business applications.

We know that a program is a collection of data and code. In large programs, data is typically represented using structures, and code using functions. In C, both structures and functions are defined in the global scope. The problem arises when the number of structures and functions increases—it becomes difficult to determine which structure is being processed by which function. Everything gets mixed up, leading to confusion for the programmer.
Conclusion: While the C language provides good control statements and function support, it is better suited for small projects rather than large ones

C++ was invented to support the concept of Object-Oriented Programming (OOP). You may often hear phrases like **"C++ with OOP" or "OOP with C++."** OOP is a set of principles and approaches that every programmer needs to follow. To understand OOP, let's explore it with an example like traffic systems in city.

Traffic rules and regulations guide the traveller easy to move in complex roads. In this case, traffic systems or approaches, such as signal-lights, zebra crossings, and dividers, guide travelers to move smoothly. Similarly, in OOP, there are three main principles or approaches: Encapsulation, Polymorphism, and Inheritance. These principles guides or forces the developers to develop application in well planned manner. These systematic approaches were developed to help designers as well as programmers to write complex programs easily. These integrate the design and development of code, making it easier for programmers.

As there are no specific approaches for writing programs in a systematic way, which can lead to every one writing in their own style, creating complexity for others. In OOP, when these approaches are followed, everyone codes in a **uniform style,** making the code easier for others to understand and work with.
As it promotes **uniform coding**, if someone develops a program in OOP style, others can easily understand it.

This uniform style makes OOP code easier to read and understand, even in large projects. Because of this, C++ with OOP is suitable for both small and large programs. OOP is widely accepted because it makes complex programs easier to design, develop, understand, debug, modify, and extend.

As a C programmer, you may think, "C++ is an advanced language compared to C." However, that is not entirely accurate. It is not an advanced version of C, but rather an extension of it. C with oops == C++. That's why it's called C++ (like an increment of C's value).

Although an in-depth discussion of object-oriented programming (OOP) concepts will be presented in the middle of this book, these concepts are gradually and implicitly introduced from the very beginning.

C++ is a superset or extension of C, meaning all features available in C are supported or adopted by C++. Additional features have been added to support object-oriented programming (OOP). C++ adopts all the basic elements from C, such as data types, operators, if statement, if-else statement, loops, arrays, functions, pointers, strings, structures, etc. Even the syntax of these elements has not been altered. Therefore, all C programs can also be compiled and executed on a C++ compiler. Remember C++ is a superset of C.

It's important to note that before diving into C++, you should have a solid understanding of functions, pointers, and structures in the C language. Remember, C++ is an extension of C, and your goal is to learn the additional features of C++. Once you are familiar with C, learning C++ will become more interesting.

This book is designed for those who are proficient in C and want to learn C++. Its focus is on explaining the internal workings of object-oriented programming with a limited number of examples. The examples provided are relevant to everyday life, presented in a straightforward way, and the same examples are repeated to help readers focus on concepts rather than on programs.

Data Types in C++

In addition to the data types available in C, C++ introduces three additional data types: long long, wchar_t, and bool. Data type sizes differ from compiler to compiler based on the operating system.

Data Type	Description	Size (Typical)
short	Short integer value	2 bytes
int	Integer value	4 bytes
long	Long integer value	4 or 8 bytes
long long	Long long integer value	8 bytes
float	Single-precision floating point	4 bytes
double	Double-precision floating point	8 bytes
long double	Extended-precision floating point	8 or 12 , 16 bytes
char	Single character	1 byte
wchar_t	wide character	2 or 4 bytes (uni-code)
bool	Boolean value	1 byte (true/false)

Operators

C++ provides some additional operators extension to the operators in C. The scope resolution operator (::), the reference operator (&), the new and delete operators, the i/o insertion-extractor operator (<<,>>), typeid, explicit, etc. Let us revise some important and very common operators.

Arithmetic Operators	+ - * / %
Relational Operators	< > <= >= == !=
Logical operators	&& || !
Bitwise Operators	& | ^ ~ << >>
Unary Operators	+ (+ve sign) - (-ve sign) ++ -- & *
Ternary operator	? :
Assignment Operators	= += -= *= /= %= &= != ^= <<= >>=
Comma operator (,)	**Example: a = (b = 3, b + 2);**
sizeof	**Example: sizeof(int), sizeof(float)**

Operator precedence (like BODMAS rules in maths)

In general, complex expressions are formed by combining mathematical or logical operators. While evaluating such expressions, sub-expressions are initially evaluated according to their precedence. For example, the expression 5 + 2 * 4 is evaluated as 5 + 8 → 13. Observe the following table showing the relative precedence of operators in C. Just take a look at this precedence; we will learn about it in detail in the next chapters

Priority 1	( ) [ ] -> .
Priority 2	! ~ +(sign) -(sign) ++ -- (pre incr/decr)
Priority 3	sizeof &(reference) *(de-reference)
Priority 4	* / %
Priority 5	+ -
Priority 6	<< >>
Priority 7	< <= > >=
Priority 8	== !=
Priority 9	^ | & (ORing, ANDing)
Priority 10	&& ||
Priority 11	?: (conditional operator)
Priority 12	= *= /= %= += -= &=
Priority 13	^= |= <<= >>=
Priority 14	Comma operator (,)
Priority 15	++ -- (post increment/ decrement)

In the above table, Operators in the same row have the same precedence. Among same precedence operators, the evaluation takes place from left side to right side in the expression. Only the assignment & unary operators are performed from right-to-left. eg: a=b=c=d, here first c=d, then b=c , and finally a=b;

Comma operator(,)

It is used to separate two or more instructions in the program. It is used in several contexts in the programming. The actual behavior of comma operator is that, it returns the right hand side operand value as the result of the expression.

```
a=2, b=3, c=10;    // Here comma works as separator
c=a , b;           // it is combination of two instructions "c=a  and b",  but 'b' does nothing
c = (a,b);         // it is also a combination of two instructions,   "a and c=b"
```

C++ keywords

Category	Keywords
Data Types	int, char, float, double, void, bool
Modifiers	long, short, signed, unsigned
Control Flow	if, else, switch, case, default, for, while, do, break, continue, return, goto
Functions & Exception Handling	void, return, throw, try, catch
Memory Management	new, delete
Access Modifiers	public, private, protected
Classes & Inheritance	class, struct, union, this, virtual, friend, typename, enum
Operators	sizeof, typeid, const_cast, dynamic_cast, reinterpret_cast, static_cast, explicit
Namespaces	namespace, using
Templates	template, typename
Miscellaneous	const, volatile, static, extern, inline, typedef, register, mutable, constexpr, asm, true, false
C++11 and Later Keywords	override, final, nullptr, decltype, noexcept, constexpr, static_assert, thread_local

true and false keywords

In C, a conditional expression evaluates to either a true or false Boolean value. Any nonzero value, including negative numbers, is considered true, while 0 is considered false. In C++, these Boolean values (0 and 1) are represented by the keywords 'true' and 'false'. The value 0 is automatically converted to false, and any nonzero value is automatically converted to true. The reverse is also true: true is converted to 1, and false is converted to 0. Additionally, C++ introduces a Boolean data type called 'bool', which can only have the values true and false.

Therefore, in C++, the expression that controls a conditional statement is technically of type bool.

```
int   main()
{       bool  k = false;
        if( k==false)
        {       k=true;
                k=0;            //  this converted as k=false
                k=1;            //  this converted as k=true
                k=10;           //  this converted as k=true
                k=10-10;        //  this converted as k=false
                k=-10;          //  this converted as k=true
                -----
        }
        k=true;
        if( k==1)       //  if( k==true)
        {       ------
                ------
        }
}
```

Tokens

Consider the expression X + Y. This expression consists of three tokens: X, Y, and +. Here, X and Y are called operands, while + is called an operator. The compiler breaks down all instructions into individual tokens to check for syntax errors. This process of splitting expressions into tokens and checking them is known as parsing. A token can be defined as a fundamental element in a program that is parsed by the compiler. Tokens can include keywords, operators, identifiers, constants, or other symbols. For example

```
a+b         //  this expression has 3 tokens: a , b , +
a<=b        //  this has 3 tokens 'a', 'b', '<='   ( here '<=' is a single token )
c++         //  this has 2 tokens  c , ++         ( here '++' is a single token )
if(a<b)     //  this has 6 tokens
```

note: the symbol '+' is different from '++' , both are different operators.

Initialization VS assignment of variable

We can set a value to the variable at the time of declaration ie, at the time of creation of memory. This is called initialization of variable. Assigning a value after declaration is said to be assignment.

```
eg1:  int K1=10; // this is called initialization, not an assignment, here k1 will  be initialized with 10;
eg2:  int K2;
        ----
        K2=20;          // this is not initialization, this is called assignment.
```

Note: In the above two examples, initialization and assignment result in the same outcome, so either approach can be followed. However, in case of **arrays**, **structures**, and **strings**, the syntax for **initialization** differs from that of **assignment**.

C++ comment line

Comment lines are meaningful passages of text that provide information about the author of the program, purpose of program, and other relevant details. It is a good programming practice to add comments to increase the readability of the program. Programmers typically add comments at complex instructions to make the logic easier to understand. Comments can be added anywhere in the program. C++ supports single line comment.

syntax: // ……..comment line…….

example:
```
// this program is written by Srihari, dated on 10-9-2024
// this program calculates the salary of employee
```

C++ is a superset of C, so we can use both C and C++ comment styles in C++ programs
The C comments can be used for single line as well as multiple lines

```
/* ……. comment line …… */      → as single line comment

/* ……. comment line 1 ……       → multiple lines comment
   …… comment line 2 ……
   …… comment line 3 …… */
```

These comment lines are ignored (skipped) during the compilation of a program and are not executed.
Comments are not part of the program, used for documentation purposes.
C comments can be used for both single and multiple lines, while C++ supports only single line.

Coding style of C++ program

Unlike some other programming languages (COBOL, FORTRAN, etc.), C and C++ give programmers the freedom to follow their own coding style. However, ensure that, the program should be readable and easy to debug by others. For example, see the following instructions

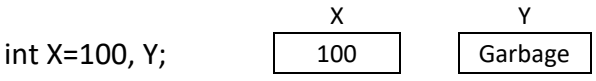
```
a=10;
b=20;
c=a+b;
```

the above sequence of instructions can be written in single line as: a=10; b=20; c=a+b;
to increase readability, we can give more spaces between expressions, that is, we can add extra spaces before and after of every token such as punctuation symbols, operators, variables and other places.
For example:

1. a+b → can be written as: a + b // observe spaces added before & after '+' symbol
2. a+b<d*e → can be written as: a + b < d * e // observe space added here
3. printf("hello"); → can be written as: printf ("hello") ; // these are 5 tokens

What is Garbage value?

At the time of declaration, if a variable is not initialized with any value, it holds an unknown value by default, called a garbage value. This value can be positive, negative, or sometimes zero.



Here X contains 100, but Y contains garbage value.
After a program finishes execution, its binary code or data is not immediately erased/cleaned from memory by the operating system. It remains in memory until another program is loaded into the same space.

When a new program loads into this memory, the old program's code and data are simply overwritten. However, when memory is allocated for variables in the new program, the locations may still contain leftover binary values from the previous program. These leftover values are considered **garbage** by the new program. Therefore in the above X,Y; the variable 'Y' contains a garbage value.

Sometimes, this garbage may even originate from the same program. For example, when a function terminates, its local variables' memory space is freed and can later be reallocated for other variables in another function.

Operating System

The term O.S refers to a set of programs that manage the resources of a computer. The resources of a computer include the processor, main memory, disks, and other devices such as the keyboard, monitor, and printer that are connected to it. The operating system also provides a user-friendly interface, allowing users to operate the computer without needing to know the details of the hardware. Thus, the interface conceals the underlying workings of the computer. The main functions of an operating system include memory management, process management, disk management, I/O management, security, and providing an interface for the user, among others.

The most widely used operating systems are MS-DOS, UNIX, and WINDOWS. DOS is a simple operating system that was primarily used on PCs in the past. UNIX, Linux, Mac, and Windows are used across a variety of computers, including mainframes, servers, graphics workstations, supercomputers, and also PCs. Nowadays, Windows and Linux are primarily used in workstations and PCs.

When a user runs a program (app) on the computer, the operating system loads the program into main memory (RAM) from the hard disk and then executes it instruction by instruction with the help of the processor. The processor can take and execute only one instruction at a time. Thus, this loading and execution of instructions is done under the control of the OS. The OS is primarily responsible for managing these tasks and also ensures that the computer can perform other operations.

The relation between hardware, operating system, user-program, and user can simulate with a banking-system such as bank, employee, transaction and customer. The hardware as a bank and O.S as a bank employee, who works dedicatedly to organize all transactions of a bank, whereas the user as a customer and user-program as a transaction, the user submits his program just by giving a command to the computer; the responder, the O.S, takes up the program into main memory and process the instructions on the hardware under its control. The following picture shows the layers of interaction between user and the computer.

User
User program
Operating system
Computer hardwar

Customer
Transaction
Bank employee
Bank

In this way operating system is the total responsible for every task which is performed in the computer. The world biggest software are nothing but operating systems like windows, unix, anroid, linux.

about main() function

In C and C++, the main() function is the entry point of the program. The return type of main() is typically int, and this is the standard that most modern compilers adhere to. For example

```
int  main()                          int  main()
{       ------------                 {     ------------
        ------------                       ------------
        return 0;                          return 4;     // here 4 is the error-code
}                                    }
```

here returning '0' indicates that the program has executed successfully, while returning a non-zero value indicates an error. The zero is returned to the operating system, telling that the program ran with zero errors (no errors). The exit(0) or return 0 at main() fn tells to the operating system to terminate the resources allocated to the program like closing files, closing printer buffers, closing IO buffers, deleting dynamic memory, etc.

This return value is stored in the operating system log files (history files) for further reference or feedback. In C, we have used void main() instead of int main() in simple programs and educational programs, where the return value is not considered important. However in critical system programing the "int main()" is the mandatory.

Compiling and Running programs

Generally, when we talk about Java or Python, there is typically a single compiler used worldwide. However, when it comes to C or C++, multiple compilers are available. You might wonder why this is the case. The reason is that C and C++ are often used for system programming (hard level programming), and many hardware and software companies have customized the compiler to meet their specific requirements. Notice that, Java or python does not support system programming, only supports application programming.
Some famous compilers are: GCC (GNU Compiler Collection) , Clang , MSVC (Microsoft Visual C++), intel C++ Compiler, MinGW (Minimalist GNU for Windows), **Cygwin,** Embarcadero C++Builder, Tiny C Compiler (TCC), **Comeau C/C++,** Borland C++ Compiler, PGI (NVIDIA HPC Compiler), **IBM XL C/C++ Compiler.**
As a student, we use GCC compiler, and this supports all operating systems.
The **C++** programming language was initially standardized in 1998, the evolution versions are C98, C++03, C++11, C++14, C++17, C++20, and C++23.
The famous C++ IDE are: Code Blocks, Eclipse, Visual studio, Borland Turbo C++(old), Code Lite, Net Beans, Dev C++. The programs in this book are tested under CodeBlocks IDE with GCC compiler.

The CodeBlocks IDE is available along with cpp compiler in the internet, google throw the keywords: "Download codeblocks-10.05mingw-setup.exe" , the best download site is: https://sourceforge.net.
The C++ compiler can also function as a C compiler. If you save your program with the file extension '.c', it will be compiled as a C program. If you save it with the extension '.cpp', it will be compiled as a C++ program.

We can use C language libraries in C++, such as printf(), scanf(), pow(), and sqrt(). In C++, there are special I/O functions for reading values from the keyboard and printing values on the screen. These I/O functions are introduced in the middle of this book; until then, we have used printf() and scanf() because they are often considered more sophisticated than the standard C++ I/O.

Type casting

Process of converting a value from one type to another type is called as type casting. In C, the type casting is done in two ways. ①**Implicit type casting (automatic casting)** ②**Explicit type casting (manual casting)**

①Implicit Type Casting

For example, in the expression 10 + 20.54 (int + float), the value 10 is automatically converted from an integer to a float (10 to 10.00). This is because we must hand over the same size and type of values to the processor before computing. The processor performs operations bit by bit on two values, so they need to be of the same size and type. The compiler automatically handles this conversion when the types in the expression do not match. In this way, the compiler promotes lower-type values to higher-type values within the expression.

Expression →	**After conversion →**	**Result**
int/int	no conversion	int
float/int	float/float	float
int * int	no conversion	int
int * long int	long int * long int	long int
int * float	float * float	float
float * double	double * double	double

Let us see some examples,

eg1: 10.5 + 20; // here the value 20 will be converted into 20.00 by the compiler implicitly

eg2: 10 * 20.00; // here the value 10 will be converted into 10.00 by the compiler implicitly

eg3:
```
int  x = 10;
float y;
y = x;          // Here 'x' value 10 will be promoted to 10.00  and stored into 'y'
```

eg4:
```
int x;
float y=29.34;
x=y;        // Here the integer part of 'y' value 29 is assigned to 'x' ( fraction bits will be truncated/omitted )
            // some C++ compilers does not allow this direct assignments where we need written as: x=(int)y;
```

eg5:
```
long int Y = 9023455L;
int  X;
X = Y;   // here we are trying to assign 4-byte value into 2-byte memory, so only right most 2byte value of Y
             is assigned to X, so results loss of some bits.  (so proper value will not be stored in X)
         // some c++ compilers does not allow this type of direct assignments where we need to write as X=(int)Y;
```

eg6:
```
void main()
{     float k;
      int x=10, y=3;
      k=5+x/y;
      printf("\n Result of K = %.2f " , k );    // Result of K=8.00
}
```

In this expression, x/y gives an integer result of 3 (not 3.33) because both x and y are of type int. This result is then added to 5, which is also an integer, and the final result is implicitly converted to a float before being assigned to k. Therefore, the output is: Result of K = 8.00.

To get the exact value of the division, explicit casting is the solution, as shown in the examples below.

② Explicit type casting

Sometimes, we need to convert explicitly to get desired result from the expression.

syntax: (conversion-type) expression

eg1: (float) 15 → 15.00

(int) 2.6 → 2

In the above expression, 15 is **int**. But, upon prefixing **(float)15,** its type will be changed to float as 15.00

eg2:
```
void main()
{   int  x=10, y=3;
    float k;
    k = 5 + x/y;
    printf("\n Before   type  casting K=%f " , k );
    k = 5 + (float) x/y+5;   // here x is converting by us,  whereas y will be converted by the compiler.
    printf("\n After type casting K= %f ", k );
}
```

Before type casting K=8.000000

After type casting K= 8.333333

eg3:
```
void main()
{   int X=30, Y=20000;
    long int k;
    k = 10 + X*Y;
    printf("\n Before  type  casting K=%ld", k);
    k = 10 + (long int) X*Y;
    printf("\n After type casting k=%ld", k);
}
```

Before type casting K=10186 (un-expected value)

After type casting K= 600010

In the expression K=10+X*Y, the product of X*Y is first calculated and stored in a temporary variable before being added to 10. This temporary variable is a nameless variable automatically created by the compiler (let's call it T). Then the instruction K=10+X*Y is executed as: T=X*Y; K=10+T;

Here, X and Y are of type int, so T will also be of type int. However, if the result of X*Y exceeds the range of an int, it cannot be stored in T, causing a loss of some bits due to overflow.

The solution is to write the expression as K = 10 + (long int) X*Y. Here, we explicitly convert the X to long int, and the compiler automatically converts b to long int. As a result, T will be long-int, which can safely store the X*Y value.

Changing a lower type to a higher type is called *promotion (expanding)*, while the reverse is called *demotion (narrowing)*.

Representing constants in C++

In programming, data is represented in two ways: as constants and variables. Constants are represented in a special way by adding format strings or other symbols to their values. For example, 10L represents a long integer, and 10F represents a float.

The compiler automatically recognizes certain types of constants in the program. For example, the constant 10 is considered an int-type, 34.55 as a double-type, and 'A' as a char-type. These are default types automatically understood by the compiler. However, some constant types need to be specified explicitly using format strings. The following list explains the different types of constants. Avoid using symbols like commas, spaces, or quotation marks when specifying constants.

integer constant (int constant)

A collection of one or more digits with or without a sign referred to as singed int constants. This is the default type for integral values. eg. 5, 256, 9113, 6284, +25, -62, etc are valid signed integers.

Similarly, to represent unsigned int, the format string ‘u’ or ‘U’ is suffixed to the value.
eg. 67U, 43U, 4399u, 45u are valid unsigned integers.

Note that, -3428u is also a valid number. Here this -3428 is converted into equivalent unsigned integer to 62108. Because, here the sign bit is also considered as data bit. (But this style is not recommended)
Some **invalid** declaration of integer constants is:

15,467 $25,566 4546Rs

long integer constants

for the long integer constants, the format string ‘l’ or ‘L’ is suffixed to the value.
eg: 2486384L –123456L 5434545l –34545l are valid long int constants
893434lu -3Lu … etc are valid unsigned long integers.

Octal int/long integer constants

for the octal integers, zero is prefixed before the value. (0 is like octal).
eg: 0123, 0574, 01, 046342L etc are valid octal integers.
0181, 09, 12, etc are in-valid octal integers. (In octal system 0-7 digits are used)

Hexadecimal int/long integer constants

in the hexadecimal number system, the symbols 0, 1, 2, 3 … 9, A, B, C, D, E, F are used.
(Total 16 symbols are used; because, the number system’s base is 16)
The symbol “0x or 0X” is prefixed before the hexadecimal number.
For example: 0x1, 0x123, 0xa5, 0xfldb, 0x0, 0xABC, 0xflc etc are valid hexadecimals.

character constants

To specify a character constant, we must enclose the character within the pair of single quotes.
eg. ‘A’ ‘z’ ‘s’ ‘8’ ‘+’ ‘;’ etc

string constants

It is a collection of characters enclosed within double quotes, used to represent names, codes, abels, etc.
eg: “Hello! Good morning!” “Magic mania” “A” “123” “A1” “#40-5-8A” “C-Family”

float constants (Real numbers)

We can specify the real numbers in two different notations, namely general and scientific notation.
Here the format string 'f' is attached at the end of value.
In normal representation: 3.1412f -25.62f 98.12f -045632.0f 1.f 9.13f
In scientific exponential representation: the syntax is [-]d.dddde[+/-]ddd where d is any digit.

2 . 32e5f (means 2.32 x 10^5 i.e. 232000.0)
12 . 102e-3f (means 12.102 x 10^-3 i.e., 0.012102)
-2 . 165e6f (means -2.165 x 10^6 i.e., -2165000.0)
3 . 68e3f (means 3.68 x 10^3 i.e., 3680.0)

double constants: **This is the default type for real numbers (floating point values).**

eg: 1.2 45.3 4.39393 349.34899034
45.4e15 44.54e4 ('l' or 'L' is option for double)

long double: **for the long double, the format string 'l' or 'L' is used.**

eg: 3.14L
314.41 (Equivalent to 314. 0L)
3.68e3L (Means 3.68 x 103 i.e., 3680.0L)

Migrating from C to C++

The following example illustrates why we should migrate from C to C++. Let's calculate the area and perimeter of a circle with a given radius. The two examples below demonstrate how to find these values for a single circle and for two circles. C-language beginner might write the code in the following style

Code to find single Circle object

```
int  main()
{   float radius, area, perimeter;
    printf("enter radius::");
    scanf("%f", &radius);
    area=3.14*radius*radius;
    perimeter=2*3.14*radius;
    printf("area is %f ", area);
    printf(" perimeter is %f ", perimeter );
}
```

Code to find two Circle objects

```
int main()
{   float radius1, radius2,  area1, area2,  peri1, peri2;
    printf("enter radius1 & radius2::");
    scanf("%f%f", &radius1, &radius2);
    area1=3.14*radius1*radius1;
    area2=3.14*radius2*radius2;
    ------------
    ------------
}
```

The first program works fine when we want to find the area and perimeter for a single circle. However, when we want to find the values for two circles, as shown in the second example, the code becomes complex and harder to read. If we need to find for three circles or an array of circles, the code becomes even more complicated and unreadable.

When the number of input circle objects changes, the entire code needs to be modified. This means that the code has to be rewritten whenever the input data size changes. However, by using functions, pointers, arrays, and structures, we can simplify the code. Once the code is written using these features, it doesn't need to be changed, regardless of the number of input circle objects. This allows the code to remain flexible and scalable without requiring modifications as the number of circle objects increases.

Here, the code is divided into two parts. Part1 defines how to process a single circle object, and this code can be reused for any number of circles. Part2 defines the main() function, here Part1 is used based on the input size. This is shown below.

Part1

```
struct Circle
{   float radius, area, perimeter;
};
void  scan( struct Circle *p)
{   printf("enter radius ::");
    scanf("%f", &p->radius);
}
void  find( struct Circle *p)
{   p->area = 3.14 * p->radius * p->radius;
    p->perimeter = 2* 3.14 * p->radius;
}
void  print( struct Circle *p)
{   printf("area is %f", p->area);
    printf("perimeter is %f", p->perimeter);
}
```

Part2

The main() fn for calculating single circle

```
int  main()
{   struct Circle  k;
    scan( &k );
    find( &k );
    print( &k );
}
```

The main() fn for calculating two Circles data.

```
int  main()
{   struct Circle  k1, k2;
    scan( &k1 );   scan( &k2 );
    find( &k1 );   find( &k2 );
    print( &k1);   print( &k2 );
}
```

You might have observed the above code that, even though the number of input circle objects changes, the part1 code remains the same without any modifications. Generally, in a development environment, part1 is written by some 'X' person and part2 code by some 'Y' person(user). part1 code defines generic code which works for single or multiple objects.

Here part1 code gives an idea how to write an independent, reusable, modifiable and expandable code without affecting the part2 code. Part1 code works like API/Libraries in our project. In this way, using **functions, pointers and structures**, we can write independent simplified reusable code. Part1 of the code is created by senior industry experts, while part2 is developed by the next group of developers according to the project's requirements. The part2 people are also called junior developers or users.

Comparing two Dates

Let us have one more example: scanning two dates and finding both are equal or not

```
int main()
{   int d1,m1,y1, d2,m2,y2;
    printf("enter date1::");
    scanf("%d%d%d", &d1, &m1, &y1);
    printf("enter date2::");
    scanf("%d%d%d", &d2, &m2, &y2);
    if( d1==d2 && m1==m2 && y1==y2 )
          printf("equal");
    else  printf("not equal");
}
```

Here, the code is fine for comparing two dates, but if there are three or more input dates, the code needs to be changed, and it becomes more complex. Like above program, if we use functions, pointer and structures then we get following simplified code

Part1

```
struct Date
{   int d, m, y;
};

void  scan( struct  Date  *p )
{   printf("enter date ::");
    scanf("%d%d%d", &p->d, &p->m, &p->y);
}

int  compare( struct  Date  p  ,  struct  Date q )
{   if( p.d==q.d && p.m==q.m && p.y==q.y)
          return 1;
    else   return 0;
}
```

Part2

```
Comparing 2 dates
int main()
{   struct Date  a , b;
    scan( &a );
    scan( &b );
    k=compare( a , b );
    if ( k == 1)   printf(" equal");
    else  printf(" not equal");
}
-----------------------------------------
Comparing 3 dates
    struct Date  a, b, c;
    ----------
    k = compare( a , b );
    if( k==1)      // if a,b are equal then compare b,c
         k=compare( b , c );
    if ( k == 1)  printf("all are equal ");
    else  printf(" all are not equal ");
```

Again, in the above code you might have observed that, the part1 code remains unchanged regardless of whether the number of dates to be compared is 2 or 3. This demonstrates how we can write independent and reusable code using functions.

Adding two times

Let's consider an example of finding the addition of two times, such as the total hours worked by an employee in two shifts. We'll use functions and structures to handle this in a clean and reusable way.

Part1 code

```
struct Time
{   int d, m, y;
};
struct Time scan()
{   struct Time k;
    printf("enter time :");
    scanf("%d%d%d", &k.h, &k.m, &k.s);
    return k;   // returning scanned time(three values)
 }
struct Time add( struct Time  a , struct Time b)
 {  long int  k;
    struct Time  t;
    k=(a.h+b.h)*3600 + (a.m+b.m)*60 + (a.s+b.s);
    t.h=k/3600;
    t.m=(k%3600)/60;
    t.s=(k%3600)%60;
    return t;
}
void write( struct Time k)
{   printf(" time is: %d-%d-%d", k.h, k.m, k.y );
}
```

Part2 Code

```
// adding two shifts time
int main()
{   struct Time a, b, c;
    a=scan();
    b=scan()
    c=add(a,b);
    write(c);
}
-----------------------------------------------
// adding three shifts time
int main()
{   struct Time a, b, c, d
    a=scan();
    b=scan();
    c=scan();
    d=add(a,b);    // d=add( a , add(b,c)  );
    d=add(c,d);
    write(d);
}
```

Yes, by observing the above programs, it's clear that using functions, pointers, structures significantly improves code independency, simplicity, reusability, and modifiability. This approach makes it easier to manage, understand, and extend the code. Let us migrate to C++;

C++ is a superset of C, meaning it was built upon C language features. C++ is not an advanced language; it is merely an extension of C. All the concepts of C can be used in C++ without any thought, such as data types, control structures, arrays, functions, pointers, structures, etc.

All C programs can be compiled using a C++ compiler without any modifications. In C++, extra features were added without disturbing existing C language features. For example, bool as a data type, new/delete for dynamic memory allocation, and cin/cout for input/output, etc. We will discuss C++ features in the upcoming chapters.

In C++, we use the 'class' data type in place of 'struct' data type. The 'class' is more sophisticated than struct. In structures, we define only data members, but their manipulation functions should be written outside the structure block. This is because the structure syntax was designed in that way. However, in a class, both the data and its manipulation functions are written inside the class block.

The following example demonstrates addition of two values using C++. In C++, we use ‘class’ keyword in case of ‘struct’. Remember, in C++, we place data members and their manipulation functions inside the ‘class’ block.

Part1 Code

```
class Adder
{     int a, b, total;

      void  set()
      {     a=10;
            b=20;
      }

      void  find()
      {     total=a+b;
      }

      void  print()
      {     printf(“%d  %d  %d ”, a , b, total );
      }
};
```

Part2 Code

```
int  main()
{     class Adder  k;
      k . set( );
      k . find( );
      k . print( );
}
```

There's no doubt that a 'class' is more sophisticated than a structure; it offers superior syntax and additional features. In this chapter, we will briefly discuss the 'class' and the 'this' pointer. In the following chapters, we will dive deeper into classes and their syntax.

I want to reveal an important fact about the C++ compiler: every C++ program is ultimately converted into C-like code by the compiler because C is a native language that closely resembles machine code. Therefore, any programming language code is eventually converted into C-like code. Let's see how the above C++ code converts into C code (the exact code may vary).

C++ code	Code after converting C++ to C

class Adder
{ int a, b, total;
 	
void set()
{ a=10;
b=20;
}
 	
void find()
{ total=a+b;
}
 	
void print()
{ printf("%d %d %d ", a , b, total);
}
};
 	
int main()
{ class Adder k;
k . set( );
k . find( );
k . print( );
}
	struct Adder
	{ int a, b, total;
	};
	
	void set(struct Adder *this)
	{ this->a=10;
	this->b=10;
	}
	
	void find(struct Adder *this)
	{ this-> total = this->a + this->b;
	}
	
	void print(struct Adder *this)
	{ printf("%d %d %d", this->a, this->b, this->total);
	}
	
	int main()
	{ struct Adder k;
	set(&k);
	find(&k);
	print(&k);
	}

Here, the class keyword is effectively converted into the struct keyword as shown above.
The call statement k.set() is converted into set(&k).
The call statement k.find() is converted into find(&k).

In this process, '&k' is passed as an argument to the function by the compiler. Within the function body, the receiving parameter 'this' is used. The 'this' pointer is a hidden pointer created by the compiler to hold or receive '&k'. Using the 'this' pointer, the members of the structure or class are accessed as shown in the above example.

Following program finds area and perimeter of a given circle, this explains how the pointer 'this' works in the program

finding radius of circle C++ Code	C code after converting From C++

class Circle
{
float radius, area, perimeter;
 	
void scan()
{ printf("enter radius ::");
scanf("%f", &radius);
}
 	
void find()
{ area = 3.14 * radius * radius;
perimeter = 2* 3.14 * radius;
}
 	
void print()
{ printf("area is %f", area);
printf("perimeter is %f", perimeter);
}
};
 	
int main()
{ class Circle k;
k.scan( );
k.find( );
k.print( );
}
	struct Circle
	{ float radius, area, perimeter;
	};
	
	void scan(struct Circle *this)
	{ printf("enter radius ::");
	scanf("%f", &this->radius);
	}
	
	void find(struct Circle *this)
	{ this->area = 3.14 * this->radius * this->radius;
	this->perimeter = 2* 3.14 * this->radius;
	}
	
	void print(struct Circle *this)
	{ printf("area is %f", this->area);
	printf("perimeter is %f", this->perimeter);
	}
	
	int main()
	{ struct Circle k;
	scan(&k);
	find(&k);
	print(&k);
	}

The keyword 'class' is optional when using 'class' as data type.
for example, the declaration : class Circle ob1 , ob2; can be declared as Circle ob1 , ob2;
so the "class Circle" can be taken as "Circle".

Following program compares two dates and prints equal or not (observe how 'this' pointer works)

C++ code to compare two dates equality.	Code after converting 'C++' into 'C' code.

class Date
{ int d,m,y;
 	
void scan()
{ printf("enter date::");
scanf("%d%d%d", &d, &m, &y);
}
 	
int compare(class Date q)
{ if(d==q.d && m==q.m && y==q.y)
return 1;
else return 0;
}
};
 	
int main()
{ class Date a , b ;
a.scan( );
b.scan( );
k=a.compare(b);
if (k == 1) printf(" equal");
else printf(" not equal");
}
	struct Date
	{ int d, m, y;
	};
	
	void scan(struct Date *this)
	{ printf("enter date ::");
	scanf("%d%d%d", &this->d, &this->m, &this->y);
	}
	
	int compare(struct Date *this , struct Date q)
	{ if(this->d==q.d && this->m==q.m && this->y==q.y)
	return 1;
	else return 0;
	}
	
	int main()
	{ struct Date a,b;
	scan(&a);
	scan(&b);
	k=compare(&a , b);
	if (k == 1) printf(" equal");
	else printf(" not equal");
	}

following program prints multiplication table upto 10 terms

class Table
{ int n;
 	
void set(int x)
{ n=x;
}
void showTable()
{ for(int k =1; k<11; k++)
printf("\n%d*%d=%d",n, k,n*k);
}
};
int main()
{ class Table t ;
int x;
printf("enter table number:");
scanf("%d", &x);
t.set(x);
t.showTable();
}
	struct Table
	{ int n;
	};
	void set(struct Table *this , int x)
	{ this->n = x;
	}
	void showTable(struct Table *this)
	{ for(int k =1; k<11; k++)
	printf("\n %d*%d=%d", this->n, k, this->n*k);
	}
	
	int main()
	{ struct Table t;
	int x;
	printf("enter table number:");
	scanf("%d", &x);
	set(&t, x);
	showTable(&t);
	}

Finding addition of two times which is employee worked in 2 shifts

Code in C++	Code after converting C++ into 'C' code

class Time
{ int d,m,y;
 	
void scan()
{ printf("enter time :");
scanf("%d%d%d", &h, &m, &s);
}
 	
Time add(Time b)
{ long int k; Time t;
k=(h+b.h)*3600 + (m+b.m)*60 + (s+b.s);
t.h=k/3600;
t.m=(k%3600)/60;
t.s=(k%3600)%60;
return t;
}
 	
void print()
{ printf(" %d-%d-%d", h, m, y);
}
};
 	
int main()
{ class Date a , b;
a.scan();
b.scan();
c = a.add(b);
c.print();
}
	struct Time
	{ int d,m,y;
	};
	
	void scan(struct Time *this)
	{ printf("enter time :");
	scanf("%d%d%d", &this->h, &this->m, &this->s);
	}
	
	struct Time add(struct Time *this , struct Time b)
	{ int k; struct Time t;
	k=(this->h+b.h)*3600 + (this->m+b.m)*60 + ….);
	t.h=k/3600;
	t.m=(k%3600)/60;
	t.s=(k%3600)%60;
	return t;
	}
	
	void print(struct Time *this)
	{ printf(" %d-%d-%d", this->h, this->m, this->y);
	}
	
	int main()
	{ struct Date a , b;
	scan(&a);
	scan(&b);
	c = add(&a , b);
	print(&c);
	}

When one member function is called by another member function of the same class, the compiler calls it using the same object through the ' this' pointer. Let us consider the following example.

code in C++	code after converting into C-Style

class Test
{ int a,b;
void print( )
{ printf(" %d %d " , a , b);
}
void show()
{ print(); // this->show();
}
};
int main()
{ Test ob;
ob.a=100; ob.b=200;
ob.show();
}
	class Test
	{ int a,b;
	void print(Test *this)
	{ printf("%d %d" , this->a, this->b);
	}
	void show(Test *this)
	{ print(this);
	}
	};
	int main()
	{ Test ob;
	ob.a=100; ob.b=200;
	show(&ob);
	}

Finding area of rectangle

Code in C++	Fill in the code after converting it to C

class Rectangle
{ float length, breadth, area;
void set(float l, float b)
{ length=l;
breadth=b;
}
void find()
{ area=length*breadth;
}
void show()
{ printf(" area is %f ", area);
}
};
 	
int main()
{ Rectangle ob;
ob.set(4,7);
ob.find();
ob.show();
}	

Finding big of 3 numbers

Code in C++	fill in the code after converting it to C

class FindBig
{ int a, b, c;
void set(int x, int y, int z)
{ a=x; b=y; c=z;
}
int getBig()
{ if(a>b && a>c)
return a;
else if(b>c)
return b;
else
return c;
}
};
 	
int main()
{ FindBig ob;
ob.set(4, 7, 2);
int k=ob.getBig();
printf("\n big is %d", k);
}	

Finding sum of 1+2+3+4+5+…+N and 1*2*3*4*5* …*N

Code in C++	fill in the code after converting it to C

class Finder
{ int N;
void set(int a)
{ N=a;
}
int getSum()
{ int i,sum=0;
for(i=1; i<=N; i++)
sum=sum+i;
return sum;
}
int getProduct()
{ int i, product=1;
for(i=1; i<=N; i++)
product=product*i;
return product;
}
}
int main()
{ Finder ob;
ob.set(5);
printf(" sum is %d", ob. getSum());
printf(" product is %d", ob. getProduct());
}	

Code in C++	fill in the code after converting it to C

class Test
{ void show()
{ printf("\n hello");
}
void print()
{ show(); // this->show();
}
void display()
{ print(); // this->print();
}
};
 	
int main()
{ Test ob;
ob.display();
}	

Finding student result from given marks of 2 subjects

Code in C++	fill in the code after converting it to C

class Student
{ int m1,m2;
char result[30];
void set(int a, int b) →
{ m1=a; m2=b;
}
void find()
{ if(m1<50 || m2<50)
strcpy(result, "failed");
else
strcpy(result, "passed");
}
void show()
{ printf("\n %d %d %s", m1,m2,result);
}
};
int main()
{ Student ob; →
ob.set(60 , 70);
ob.find();
ob.show();
}	

How to run cpp programs: Save C++ programs with the file extension ".cpp", and do not forget to include the required header files. I have not mentioned the header files in all the examples provided in this book. We already know the commonly used files like stdio.h, math.h, process.h, iostream.h, etc.
In some modern compilers ".h" is not required. For example #include<iostream>

private vs public

Generally, variables in a class are declared as 'private', whereas functions in a class are declared as 'public'. The private members cannot be accessed outside the class—this is similar to local variables in C functions. The public members can be accessed both inside and outside the class. Default access is private.
We will discuss more about this in the next chapter. Before practicing above program in the lab, where convert all data members to private and all functions to public as given example

```
class Sample
{       private:  int  a, b;
                  float  x, y;                  // observe,  here a,b,x,y are declared as private
          public:  void show() { … }            //  show() fn,  calculate() fn are declared as public
                  void calculate() { … }
};
```

class and object

In the previous chapter, we briefly discussed about class and the 'this' pointer. Let us now look in more detail at classes and how they differ from structures. A class is a user-defined data type similar to a structure in the C; however, a structure only packs data members but not its manipulation functions. The functions must be written outside the structure block, because syntax was provided in that way. Here, data and its manipulation functions are treated separately, with each one defined independently in the global scope.

The problem with structures is that, if there are many structures and functions in a program, it becomes difficult to understand which structure's data is being processed by which function. This can lead to confusion for the programmer.

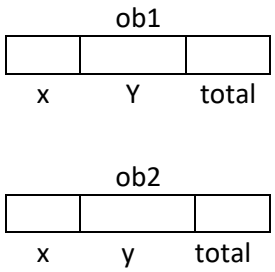
In contrast, a class encapsulates data and its manipulation functions within the class block as a single unit. It combines (packs) data and its manipulation functions under a single name. We have already seen it before. A class can be defined as a collection of data members and member functions. The following example illustrates how a structure and a class are defined in a program.

Structure in c	Class in c++
struct Adder { int x, y, total; }; void scan() { …. } void find() { …. } void print() { …. } } outside the structure	class Adder { int x, y, total; void scan() { …. } void find() { …. } void print() { …. } } inside the class };

1. The aim of a class is to provide the programmer with a tool for creating new types that can be used as conveniently as built-in types. Of course, this is similar to a structure in C, but here the data and its manipulation functions are packed and tied together as a single unit as above shown code.
2. When we define a new class in the program, it means we have added a new data type to our program. This can be used in the same way as int and float types. In the above example, the class definition '**Adder**' works as a blue-print of our-type. So the word **'Adder'** can be used like int & float types in our program.
3. Now we can declare variables of a class like: **Adder ob1, ob2;** (this is like: ***int p , q ;)***
   So, variables of a class are declared just like any other variables in the program.
4. **Class variables are called objects.** The creation of memory space for objects is the same as for structure variables in C. The following example explains how memory space for objects is created in the program.

```
Class   Adder
{       int x, y, total;
        void  scan() { …. }
        void  find() { …. }
        void  print() { …. }
};

int  main()
{       int  a,b ;              // here a,b are int-type variables
        Adder  ob1, ob2;   // here ob1,ob2  are class-type variables
        ----------
        ----------
}
```

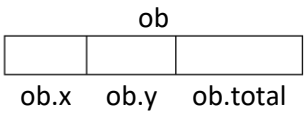
Here how ob1 & ob2 are created in memory



5. The class declaration specifies the name of the class, the list of data members, and the functions that will operate on the class data. The space for class members are created in the object, not at the time of declaration of class. Many people confuse this, so let us look at the following example.

Example

```
class Adder
{       int x=100
        int y=200;          } error,  we cannot assign values to x , y,
        int total=x+y;      } because here space will not be created for  x , y .

      void  scan() { …. }
      void  find() { …. }
      void  print() { …. }
};

int  main()
{   Adder  ob;        // here space for x,y creates in the 'ob',
    ob.x=100;         // this is not an error
    ob.y=200;         // so we can store values to x,y like structures in C.
    ---------
}
```



6. The class declaration is treated as a building plan (blueprint), whereas an object is treated as the constructed building based on that plan. The 'class vs object' is often compared to 'building plan vs building'.

7. class declaration defines the logical view at conceptual level of our type about data properties and functional behavior. Therefore it is often called template or blueprint. In contrast, an object is the physical making of a class. A class does not occupy any physical memory space in the executable file of machine code; only the object consumes physical memory in the executable. Thus, a class is considered a logical entity, whereas an object is considered a physical entity.

8. **Finally, we can conclude that a class is a data type, whereas an object is a variable of that class type.** So, a class is a concept, or model, or template that defines how an object should be.

9. We can create as many objects as we want using a class. Each object is also called an instance of the class. In programming, we first define the class body and then we create objects from it. It is similar to defining a building plan before constructing a building.

10. **Why are class variables called objects?**
    In the real world, things like a fan, pen, chair, bottle, dog, human, etc are called real time objects. Every object has its own shape and benefits. In general, an object can be defined as a collection of data properties with functionality. For example, a pen is an object with data properties like color, cost, weight, and length, and functionalities like writing, drawing, and marking. Similarly, a fan is an object with data properties like color, cost, and length, and functionalities like circulating air, clearing dust, and cooling/drying items.
    In this way, every object has its own unique shape's data properties, and functional uses. Similarly, a class object contains data and functional properties; therefore, it is called an object.

    Let us take the Date object: ob.day, ob.month, and ob.year are data properties, whereas ob.set(), ob.increment(), ob.compare() , and ob.show() are functional properties.

11. In c++, the class is the backbone, and the entire program code orbits around classes. A C program is made up of a collection of functions, whereas a C++ program is made up of a collection of classes.

Syntax of class

the syntax of a class is the same as the syntax of a structure, but we write the functions inside the class body. Additionally, access modifiers like private/public are used.

Syntax for class	Example for class

class Adder
{
private: data_member1;
data_member2;
…
member-function1() { … };
member-function2() { … };
--- 	
 	
public: data_member3;
data_member4;
…
member-function3(){ … };
member-function4(){ … };;
--- 	
};
	class Add
	{
	private:
	int x, y, total; // data-members
	public:
	void scan() // member-function-1
	{ -------
	}
	void find() // member-function-2
	{ ------
	}
	void print() // member-function-2
	{ -------
	}
	};

12. The variables that are declared inside the class-block are called **data-members** (here x, y, total) and functions that are defined inside the class block are called **member-functions**.
13. The member-functions manipulate the data-members such as scanning, processing, and printing.
14. Private members cannot be accessed outside the class-block. These are accessed only by other private and public functions of the same class.
15. In contrast, if any member need not be accessed outside the class block, we declare it as private.
16. This private access concept is same as local variable in C functions.
17. This private feature greatly reduces the complexity of the variables and also avoids misuse.
18. Private is the default access type. If no access type is specified, the compiler assumes it to be private.
19. Public members can be accessed inside & outside class block (anywhere in the program).
    This is something like global variable's access type in a C program.
20. The members of object are accessed just as structure variables in C. Here two operators are used

1. selection operator(.) 2. pointer selection operator(->)

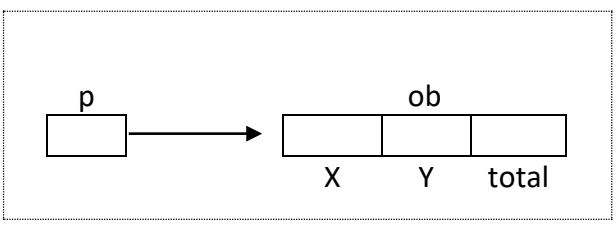
The expression 'ob.X' accesses the 'X' value in 'ob'.
The expression 'ob.Y' accesses the 'Y' value in the 'ob'.
For example filling values to 'ob' as

```
ob.X=100;
ob.Y=200;
```

If the pointer **'p'** is pointing to object **'ob'** then

```
Adder  ob , *P ;
p=&ob;
```

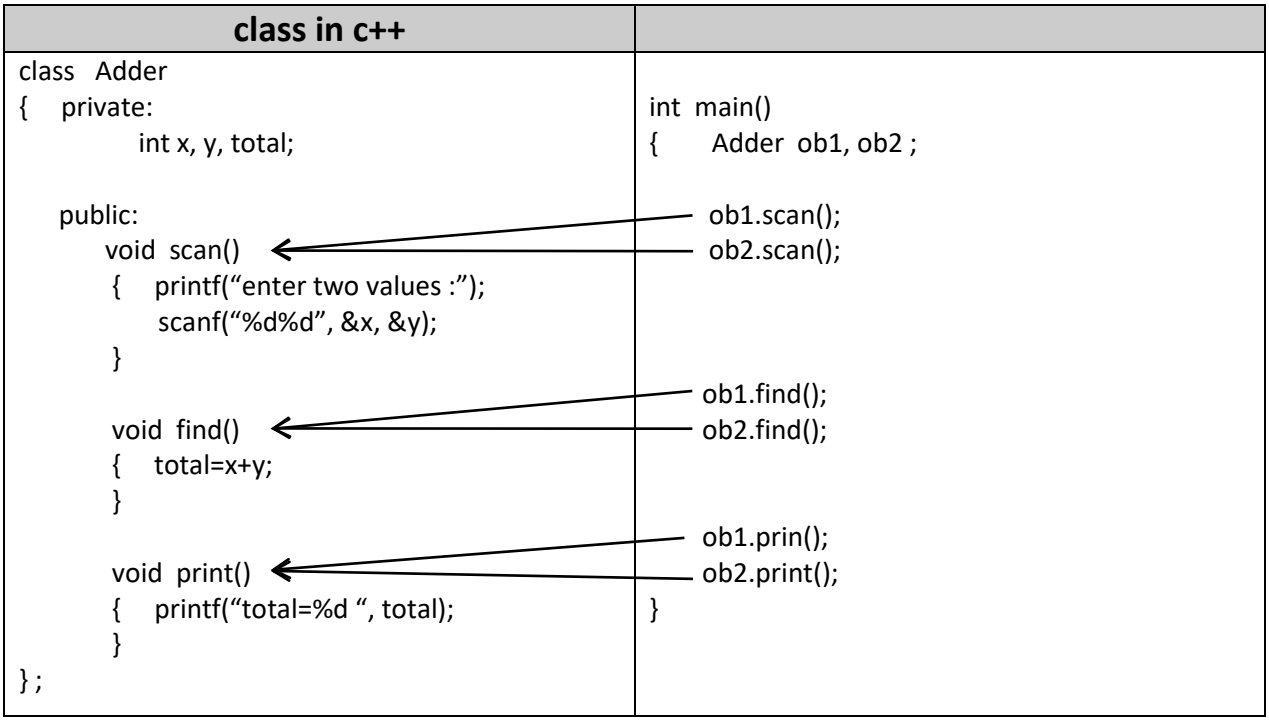
The expression 'p->X' accesses the 'X' value in 'ob'. // same as accessing structure members in C
The expression 'p->Y' accesses the 'Y' value in 'ob'.



21. The compiler uses the class definition to create memory space for objects in the executable file. Once memory space is created for all objects, the class definition is no longer required by the executable program. Therefore, the compiler doesn't allocate any space for the class definition in the machine code file. Once the building is constructed, the building plan is no longer needed unless modifications are required. Similarly, the class definition is not required by the executable program. As we know, an object is essentially a collection of data members of types like int, float, and char. Thus, the object is decomposed and managed as built-in types by the machine code.

    At the machine code level, there is no concept of classes, objects, local variables, global variables, private, public, etc. For example, we consume different dishes and tastes during meals, but they are all processed in the stomach and decomposed into carbohydrates, proteins, and other nutrients. Similarly, objects are decomposed and handled as built-in types, and eventually converted into 1s and 0s in the equivalent machine code.

22. Functions within a class are called member functions. Only one copy of each member function exists for all objects, and all objects are linked to the same functions. Many people mistakenly believe that each object contains its own set of data and set of functions. In reality, each object has its own set of data but not own set of functions; only one copy of member functions exists and is shared among all objects. This means that all objects are linked to same member functions in the program.
    The following example explains this.



The instruction 'ob1.scan()' is simply a function call with the object ob1. This is similar to calling a function in the C language. The compiler implicitly interprets this call as **scan(&ob1)**. Therefore, the address of ob1 is implicitly passed as an argument to the function and stored in a hidden pointer called 'this'. Using the 'this' pointer, the data members of ob1 are accessed as this->idno, this->m1, this->m2 within the function body. (we have already seen about the **`this`** pointer in previous chapter)

Similarly, the function call **ob2.scan()** is interpreted as **scan(&ob2)**, where the address of ob2 is passed and stored in the same this pointer. Using the 'this' pointer, the data members of ob2 are accessed.

private and public access types

Private members are not accessible outside the class, while public members can be accessed both inside and outside the class. This is similar to the concept of local and global variables in C functions. Private can only be accessed by member functions of the same class. In other words, if any variable need not to be accessed outside the class then we declare them as private. Let us see following example,

```
class  Test
{       private:  int  X;
        public:   int  Y;
        private:  void  show()
                  {   printf("%d  %d", X , Y );
                  }

        public:   void print()
                  {   printf("%d  %d", X , Y );
                  }
};
int  main()
{       Test  ob;
        ob.x = 100;      //  compile-time error,  'x' is private, not accessible outside class-block
        ob.y = 200;      //  no-error,  'y' is public, accessible anywhere
        ob.show();       //  compile-time error,  private not accessible
        ob.print();      //  no-error,  public is accessible
}
```

Here, X is private to the class scope, so it cannot be accessed in the main() function. Therefore, the statement ob.X = 100 in the main() function will result in an error.

Remember: main() fn is not member of a class so it can't access private members.

Here, Y is public, so it can be accessed anywhere in the program. The instruction ob.Y=200 at main() fn will not produce any error. Private reduces the variables complexity and also increases the security like function local variables in C. Within a class, any member function can access any other member, regardless of its public or private access modifier.

Generally, in real-world applications, most data members in a class are declared as private, while most member functions are declared as public. Typically, around 90% of data members are private, while about 90% of member functions are public. This is a common pattern in the industry, although it may sometimes vary.

Generally, data members are kept in the private section and are accessed through public interface functions of that class. This is something like consulting a manager in an office through his assistant by following some protocols. Thus private data members are accessed indirectly via public interface functions of that class. The public interface functions are typically named set() or setValue() and get() or getValue(), etc. Let us see one by one.

Points to Note: Private items are accessed through public interface functions.

---

set() and get() functions

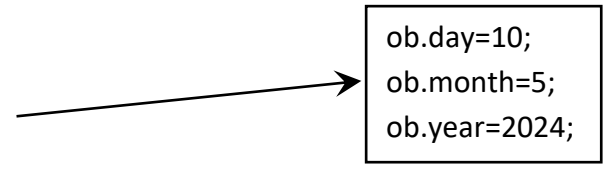
the function set() or setValue() is used to assign values to data members of an object from outside of class. These functions are typically known as "setter" methods. Following code explains how we assign values to object with and without set() function.

```
class  Date
{       public:  int  day, month, year;
};
int main()
{       Date  ob;
        ob.day=10;
        ob.month=5;
        ob.year=2024;
        printf( "\n %d - %d - %d",  ob.day,  ob.month,  ob.year );
}
```

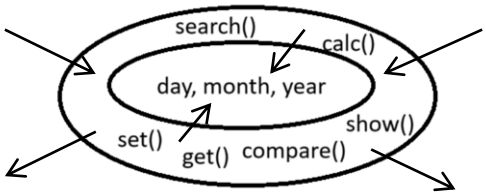
Here the object 'ob' filled with values 10, 5, 2024 and then printed using printf() statement.
We don't write the instructions ob.day = 10; ob.month = 5; ob.year = 2024 inside the main() function.
This kind of programming is not professional and puts you in hell if the program is big and many objects exist. If more objects are found, then dependency and complexity will increase. This is already discussed in the previous chapter (Migrating C to C++ chapter).

Whatever instructions that needs to be written on data members like ob.day=10, ob.moth=5, etc. These instructions must be written inside the class in member functions. We should not write the instructions outside, as this is strictly not allowed and is against to C++ programming methods. Let us see following code

```
class  Date
{       private:  int  day, month, year;
        public:   void setValues( int d, int m,  int y )
                  {       day=d;
                          month=m;
                          year=y;
                  }
                  void  printValues()
                  {       printf("%d %d %d", day, month, year );
                  }
};
int main()
{       Date  ob;
        ob.setValues( 10, 5, 2024 );
        ob.printValues();
}
```



To assign values to an object, we call the setValues() function, which takes 10, 5, and 2024 as arguments and assigns them to the data members of the object. Similarly, we print values through the printValues() function. In this way, the object's data is processed through member functions of that class. Typically, function names are like setValues(), process(), find(), compare(),print(), etc. The following picture explains how functions interact with data



setValues() with validations

The setValues() function not only sets the values to the object but also does validations before assigning values to the members. In other words, this function checks whether the values being assigned are valid or not. If they are not valid, it reports an error message to the user.

```
class  Date
{       private:  int  day, month, year;

        public:   void setValues ( int d, int m,  int y )
                  {
                        if(  d<1 || d>31 ||  m<1 || m>12 )          // checking values are valid or not
                        {       printf(" error, invalid date values ");
                                return;
                        }
                        day=d;    month=m;   year=y;
                  }
                  ------
};
int main()
{       Date  ob1, ob2;
        ob1.setValues( 10, 5, 2024);       // here we do not get any error, because valid inputs
        ob2.setValues( 40, 15, 2024 );     // here we get error message called: error, invalid date values
        -----
}
```

---

setValues() with 'this' pointer

In the above example, the names of class members are day, month, year and corresponding names of parameters of setValues() are d, m, y; Here d, m, y values are assigned into day, month, year.
But most programmers take the **same name** for both data members & parameters to avoid confusion. Here the 'this' pointer is used to differentiate these variables names, let us see the following example. Here function name setValues() is shortened as set().

```
class  Date
{   private:
        int day, month, year;
    public:
        void setValues( int d, int m, int y )
        {   day=d;
            month=m;
            year=y;
        }
        ------
};
int  main()
{    Date  ob;
     ob.set(10, 5, 2020);
     ------
}
```

→

```
class  Date
{   private:
        int day, month, year;
    public:
        void set( int day, int month, int year )
        {   this->day=day;
            this->month=year;
            this->year=year;
        }
        ------
        ------
 };
```

observe the expression: this->day=day;
already we know '**this->day'** accesses the '**ob.day'** whereas **'day'** accesses the parameter of set() fn.

get() function

In some cases, private member values need to be accessed from outside the class, where we write a get() function with public access. This function is used to get/return the values of private data members of the class. The get() function works opposite to the set() function. The set() function sets the value to the data member, whereas the get() function returns the current value of the data member. For example

```
class  Sample
{     private:  int  value;
       public:  int  get()
                {   return value;
                }
                int  set(  int  value )
                {   this->value=value;
                }
};
```

```
int  main()
{     Sample   ob;
      ob.set(70);              //  ob.value=70;
      int k = ob.get();        //  k=ob.value;
      printf(" %d ", k);   → 70
}
```

In conclusion, the set() and get() functions help us read and write the values of an object with validations. In professional programming these are used for activities such as login password checks, store or load values from data bases, session time out, etc. These functions are often referred to as setters and getters.

As data members are accessed through these public interface functions like set() and get() from outside the class, hence we place them in the private section to avoid direct access from other part of program.

Generally, if a set() function is written for a particular data member, its name should follow the format setXXX(), where XXX is the name of the member. For example: setIdno(), setName(), setMarks(), and so on. The first letter of the member name should be capitalized. Similarly, get() functions follow the same naming convention as set() function. Let us see one example

```
class   Student
{   private:   int  idno , marks;
    public:
    void  setIdno(int idno)   // to change idno of student
    {   this->idno=idno;
    }
    int  getIdno( )              // returns idno of student
    {   return idno;
    }
    void  setMarks( int marks )
    {   this->marks=marks;
    }
    int  getMarks( )          // returns marks of student
    {   return marks;
    }
};
```

```
#include<stdio.h>
int   main()
{     Student   ob;
      ob.setIdno(10);
      ob.setName("Srihari");
      printf("%d %s", ob.getIdno(), ob.getMarks() );
}
```

member vs non-member function

Member functions are part of a class and are called using an object of that class. In contrast, non-member functions are defined outside the class, in the global scope, and can be called directly, just like functions in C. These are nothing but regular C language functions.

```
class Sample
{       public:
        void   show()              // member-function
        {      printf("hello");
        }
};
void  show()                       // non-member-function ( it is a C language function )
{       printf("world");
}
int  main()
{       Sample  ob;
        ob.show();                 // hello
        show();                    // world
}
```

---

new vs old syntax of class

In the new syntax, member function bodies are defined inside the class block. In contrast, in the old syntax, prototypes of member functions were declared inside the class, but their bodies were defined outside the class. This older approach separated the design (declaration) from the implementation (definition).

New Style

```
class Sample
{       int a,b;
        public:
        void set()
        {    a=10,
             b=20;
        }
        void print()
        {    printf("%d  %d", a , b);
        }
};

int main()
{     Sample ob;
      ob.set();
      ob.print();
}
```

In this book, we followed new-style coding.

Old Style

```
class Sample
{       int a,b;
        public:
        void  set();      // proto-types of member functions inside class
        void  print();
};
void  Sample::set()   // member functions bodies outside class
{       a=10, b=20;
}
void Sample::print()
{       printf("%d   %d", a , b);
}
int main()
{     Sample ob;
      ob.set();
      ob.print();
}
```

Sometimes class and its functions body used to write in separate files. Let us following example

Filename: "sample.h"

```
class Sample
{     int a,b;
      public:
      void  set();          // functions proto-types
      void  print();
};
```

filename: "sample.cpp"

```
#include "sample.h"
void  Sample::set()
{    a=10, b=20;
}
void Sample::print()
{    printf("%d   %d", a , b);
}
```

file name: "main.cpp"

```
#include "sample.cpp"
 int main()
 {     Sample ob;
       ob.set();
       ob.print();
 }
```

Conclusion on private and public

As per C++ programming style, the data and its manipulation functions should be written inside class block. The code which performs operations on class data such as scanning, processing, and printing must be written inside class. That is, no piece of code should be written outside class, thus making the data and functionality a single unit as one object.

We know, the data members are accessed/managed through member functions of that class, for example, set(), get(), process(), print(), etc. So the data members need not be accessed directly from outside class like ob.day, ob.month, ob.year at main() fn. So these members are kept in private section as they need not be accessed outside class. This private ness makes the security (avoids misuse), and also reduces variables complexity and pollution.

Taking data members as public or private is up to you. This is just an access modifier for our safety and not a compulsion. Some member functions can be private as they need not be accessed from outside class. These private functions are accessed by only others functions of same class.

Some data members like global variables, global constants, static members, are declared mostly in public. Most of the member functions are declared in public as they have to call from outside class. These works as control interface to the class data. (That is, we handle or control objects through these functions).

In development environments, many people may involve while coding projects. Generally, total project is divided into several modules, where each module is made up of with several classes. Different classes may be written by different people.

For simplicity, let's assume two people are working on our code examples provided in this book. In our code examples, let the class be written by person X, and the main() function be written by person Y. The person X hides the data members to Y by keeping in private section. From the main() function, the person Y

access/manipulate through public interface functions like set(), get(), process(), print(), etc. So, the person Y needs not to know the names and types of data members used by person X in the class.

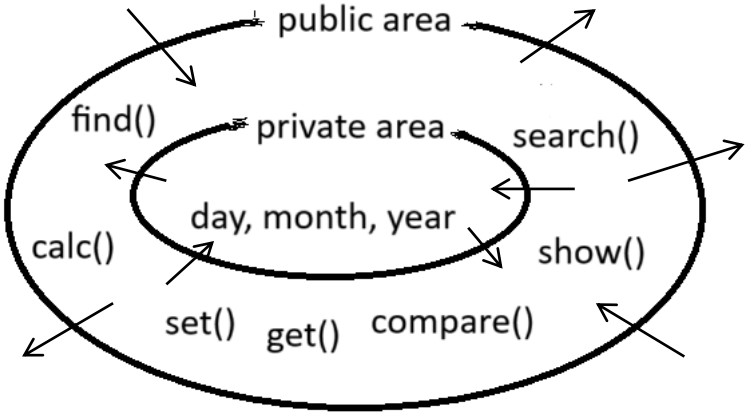
If X changes the class data members, such as by changing their names, types, or logic, there is no need to modify the code in the main() function(it does not affects the code in main() fn). Sometimes, X can add extra functionality to the class without disturbing other part of code.

The default access specifier in a class is private. That is, if we don't mention any access specifier, the compiler assumes it as private. This rule applies to both data and functions.

```
class  Sample:
{       int X;                          //  Here X  is private by default
        void showX()                    // Here function showX()  is also private by default
        {       printf(" %d  ",  X );
        }
    public:
        int Y;
        void showY()
        {       printf(" %d  %d ",  X , Y );
        }
};
```

==========

Interaction with objects: we process the object's data through public interface functions as given below pic. So, generally we keep the data in private area and functions in public area.



Let us see some sample programs, how they are constructed in C++ using classes and objects.

In this program, we set two sample input date values to object1 and object2 using the set() function in the main() function. Later, we compare the two dates and print whether they are equal.

```
#include<stdio.h>
class Date
{   int d, m, y;          // these are private by default
    public:
    void set(int d, int m, int y)
    {   this->d=d;
        this->m=m;
        this->y=y;
    }

    bool compare( Date ob2 )
    {   if(d==ob2.d&&m==ob2.m&&y==ob2.y)
            return true;
        else
            return false;
    }
};
```

```
int main()
{   Date  ob1 , ob2;
    bool  d, m, y, k;

    printf("enter date1 :");
    scanf("%d%d%d", &d, &m, &y);
    ob1.set( d , m , y );

    printf("enter date2 :");
    scanf("%d%d%d", &d, &m, &y);
    ob2.set( d , m , y );

    k=ob1.compare(ob2);
    if(k==true)
        printf("both dates are equal");
    else
        printf("both dates are not equal");
}
```

The following program adds points to a player's score. Initially, the score is set to 0.
Points are then added to the score, and the finally total is printed on the screen.

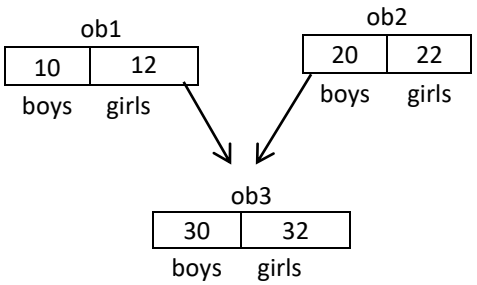
```
#include<stdio.h>
class  Player
{   int   score;
    public:

    void set( int score )
    {   this->score=score;
    }

    void addScore( int score )
    {   this->score+=score;
    }

    void show()
    {   printf("player score is %d", score );
    }
};
```

```
int   main()
{   Player ob;
    ob.set(0);            // intial score is zero
    ob.addScore(10);
    ob.addScore(20);
    ob.show();            // player score is 30
}
```

The following program finds the total count of boys & girls in two schools. The set() assigns sample input values to the objects ob1 and ob2. Later, these values are added by gender wise to ob3 and printed.

```
#include<stdio.h>
class School
{  int  boys , girls;              // default is private
   public:
   void set( int boys, int girls )
   {  this->boys=boys;
      this->girls=girls;
   }
   School add( School ob2 )
   {  School t;
      t.boys=boys+ob2.boys;
      t.girls=girls+ob2.girls;
      return t;
   }
   void show()
   {  printf("\n boys = %d", boys);
      printf("\n girls = %d", girls);
   }
};
```

```
int main()
{  School ob1,ob2,total;  //ob1,ob2 are input objects
   ob1.set(10,20);
   ob2.set(30,40);
   total=ob1.add(ob2);
   total.show();
}
```



The following program adds two times durations and prints the total time on the screen.
The two durations represent the employee's work time across two shifts in the office.

```
#include<stdio.h>
int main()
{  Time ob1, ob2, ob3;
   ob1.set(3,30,40);
   ob2.set(4,50,50);
   ob3=ob1.add(ob2);
   ob3.show();
}
```

```
Shift 1 :  3    30    40
Shift 2 :  4    50    50
       -------------------------
  total :  8    21    30
```

```
class Time
{  int h,m,s;
   public:
   void set(int h,int m,int s)
   {  this->h=h;
      this->m=m;
      this->s=s;
   }
   Time add( Time ob2)
   {  Time  t;  int  n;
      n=(h+ob2.h)*3600 + (m+ob2.m)*60 + (s+ob2.s);
      t.h=n/3600;
      t.m=(n%3600)/60;
      t.s=(n%3600)%60;
      return t;
   }

   void  show()
   {  printf("%d:%d:%d",h,m,s);
   }
};
```

The following add() function in the String class concatenates two string objects and returns the result. Initially, we assign two sample strings, 'hello' and 'world,' to object1 and object2.
We use a array char a[100] to store the input string. Let's take a look at the program below.

```
#include<stdio.h>
#include<string.h>
class String
{       char  a[100];
        public:
        void set( char str[] )
        {   strcpy( a , str );
        }
        String add( char str[] )
        {   String t;
            strcpy( t.a , a );
            strcat( t.a , str );
            return t;
        }
        String add( String ob2 )
        {   String t;
            strcpy( t.a , a );
            strcat( t.a , ob2.a );
            return t;
        }
        void show()
        {   printf("\noutput is:%s", a );
        }
};
```

```
int main()
{       String ob1,ob2,ob3;

        ob1.set("hello ");
        ob2.set("world ");
        ob3=ob1.add(ob2);
        ob3.show();             → hello world

        ob3=ob1.add("hi ");
        ob3.show();             → hello hi
  }
```

The following program adds seconds to a given time and maintains AM/PM to specify morning or evening. For example 11:40:50 PM + 2 hours → 1 : 40 : 50 : AM

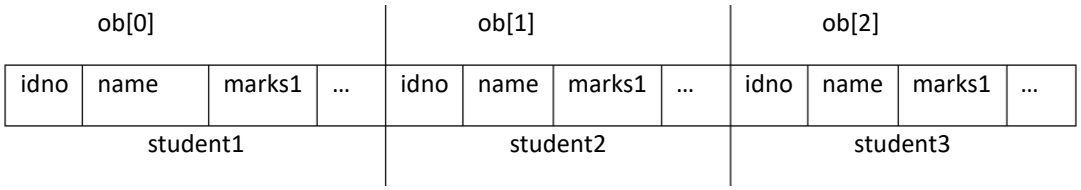
```
#include<iostream>
using namespace std;
class Time
{   int h,m,s;
    char *amPm;
    public:
    void setTime(int h, int m, int s,  char *amPm)
    {   this->h=h; this->m=m; this->s=s;
        if(*amPm=='a' || *amPm=='A')
              this->amPm="AM";
        else this->amPm="PM";
    }
    void showTime()
    {
        printf("time is %d:%d:%d", h, m, s);
    }
};
```

```
    void incrementTime(int n)
    {    int prev;
         prev=h*3600+m*60+s;  // before adding
         if( prev<=12*3600 && prev+n>12*3600 )
         {    if(*amPm=='P') amPm="AM";
              else amPm="PM";
         }
         n=n+prev;   h=n/3600;  n=n%3600;
         m=n/60;   s=n%60;
         if(h>12) h=h%12;
    }
};

int main()
{   Time ob;
    ob.setTime(11,40,50,"PM");
    ob.incrementTime(7200);  // add 2 hours to time
    ob.showTime();
}
```

Array of objects

this is same as array of structures that used in C, here every element in the array is an object which is associated with data and member functions. This is as given picture

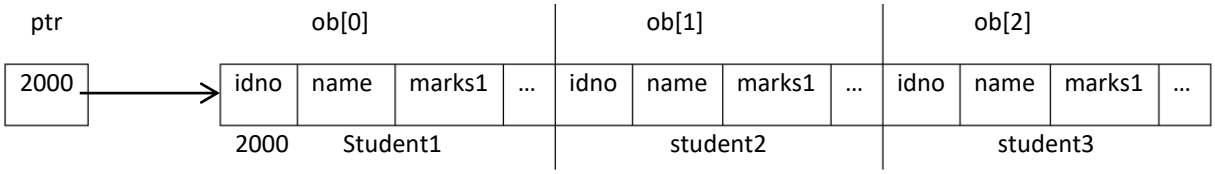
```
class Student
{   int  idno;
    char name[30];
    int marks1, marks2, total, avg;
    public:   void scan()
              {    //  scan idno, name , marks1,marks2,
              }
              void find()
              {   // find total and average
              }
              void print()
              {   //  print all members
              }
};
```




```
int  main()
{       int  i;
        Student  ob[3];            // taking array of three objects
        for( i=0;  i<3;  i++)
        {   printf("\n enter student details  %d: ", i+1);
            ob[i].scan();
        }
        printf("output  is \n");
        printf("\n idno   name   marks1   marks2   total   average");
        printf("\n  ----------------------------------------------------------------------------");
        for( i=0;  i<3;  i++)
        {   ob[i].find();
            ob[i].print();
        }
}
```

Pointer to array objects

This is same as pointer to array of structures, accessing objects using pointer is same as in array of structures using selection dot(.) and arrow(->) operator. For example, the pointer 'ptr' is pointing to array of objects as



The expression ptr[0] access the data of ob[0] // ptr[0] → ob[0]
the expression ptr[1] access the data of ob[1] // ptr[1] → ob[1]

Above main() function can be rewritten using pointer 'ptr' as

```
int  main()
{       int  i;
        Student  ob[3];             // array of three objects
        Student  *ptr;
        ptr=ob;                     //  or  ptr=&ob[0];
        for( i=0;  i<3;  i++)
        {       printf("\n enter student details  %d: ", i+1);
                ptr[i].scan();   or  (ptr+i)->scan();
        }
        (OR)
        for( i=0;  i<3;  i++)
        {       printf("\n enter student details  %d: ", i+1);
                ptr->scan();
                ptr++;          // advances to next object, but after completion of loop, the ptr moves to end of array.
        }
        ----------------
        ----------------
}
```

Function Overloading

In the C language, we can't give the same name to more than one function, where, every function carries unique function name. However, in C++, the same name can be given to two or more functions but their parameters should be different. Writing more than one function with the same name with different parameters is said to be function over-loading. Based on the arguments and parameters, the function-call statements are linked with the suitable function-body as shown below example.

```
int add( int a, int b )
{   return a+b;
}

float add( float a , float b )
{       return a+b;
}
int  main()
{   int x=add(10,20);
    printf("%d " , x );
    float y=add(10.64F , 20.54F);
    printf("%f" , y );
}
```

Here the function name is same for both functions, but their parameters are different, based on the arguments and parameters, the function-call statements are linked with the function-body.

Example2: in the following code, the function name 'add()' is used to find addition of 2 and 3 integers and also finds the float addition.

```
int add( int a, int b )
{       return a+b;
}

int add( int a, int b , int c )
{   return a+b+c;
}

float add( float a , float b )
{       return a+b;
}

int main()
{   int  x;   float y;
     x=add(10,20);
    printf("%d " , x );
    x=add(10,20,30);
    printf("%d" , x );
    y=add(10.55F, 34.34F);
}
```

note: functions are differentiated based on parameters but not on return-type.

Following example finds reverse of an integer and a string.

```
int reverse( int n )
{       int rev=0;
        while( n>0 )
        {   rev=rev*10 + n%10;
            n=n/10;
        }
        return rev;
}

char*  reverse( char *str )
{       int I , j;   char t;
        for(j=0; str[j]!='\0'; j++)      // this loop finds length of string.
        {
        }
        for( j--, i=0;   i<j;  i++, j--)    // this loop is to reverse the string;
        {   t=str[i]; str[i]=str[j];  str[j]=t;
        }
        return str;
}

int main()
{   printf("reverse of 123 is %d" ,  reverse(123) );          //  this calls first version
    printf("reverse of hello is %s", reverse("hello") );     //  this calls second version
}
```

Rules for function overloading in C++

1. All functions must have same name.
2. All functions parameters must be different as per count or types.
3. All functions must be present within a scope (sometimes in parent-class and child-class)
4. Functions are differentiated based on parameters but not on return-type.
5. **Ambiguity error:** If a function call can match more than one overloaded version equally well, the compiler will generate an ambiguity error. Overloading must be designed to avoid such conflicts. For example

#include<stdio.h>
int add(float a, float b)
{ return a+b;
}
char add(char a, char b)
{ return a+b;
}
int main()
{ int k;
k=add(65,66);
}
	Both functions are suitable for the call statement add(65,66), but the compiler is unable to determine which one to call, resulting in ambiguity. The integer values 65 and 66 can also be interpreted as the ASCII values of 'A' and 'B', which contributes to the ambiguity error. To resolve this, we need to address the ambiguity manually. For example k=add((int)65 , (int)66);
	and k=add((char)65 , (char)66);

Default arguments

Default arguments are alternative values automatically passed to a function when the user omits some arguments while calling it. These are defined in the function definition or function prototype.
Let us see the following example.

```
void show( int a , int b=20 )        // here 'b' is the default argument and its default value is 20.
{       printf("\n%d  %d" , a , b);
}
void main()
{       show( 100 , 200 );
        show( 100 );        //  here default value 20 substitutes as 2nd argument and this convert as "show(100,20)"
}
```

output: 100 200
100 20

As we know, we should pass two arguments to the show() function. However, in the second call, show(100) has only one argument; therefore, the compiler substitutes the default argument 20 as the second argument. So the call statement show(100) will be replaced as show(100, 20);

So, when arguments are omitted while calling a function, the compiler substitutes default arguments in their place. Basically, default arguments are alternative values specified during the creation of the function definition/proto-type.
For parameterized functions, the same number of arguments must be passed as the parameter count.
So, if the caller omitted such arguments, then compiler substitutes default values in their place. This feature eliminates the need for writing too many overloaded functions and also simplifies the logic and code.
These arguments must be specified from right to left in the function prototype or definition.

Example2:

```
void show( int a=10 ,  int b=20 )       // both parameters has default values
{       printf("\n%d  %d" , a , b);
}
void main()
{       show( 100 , 200 );
        show( 100);      //  this call will be replaced to:  show(100 , 20)
        show();          //  this call will be replaced to:  show(10 , 20);
}
```

output: 100 200
100 20
10 20

Let us have some valid and invalid declarations

```
void show(int a, int b=10);              // valid
void show(int a=10, int b);              // in-valid, default be given from right-to-left (without 'b' , 'a' can't be default)
void show(int a, int b=0, int c);        // in-valid, default should be given from right-to-left
void show(int a=10, int b=0, int c);     // in-valid, default should be given from right-to-left
void show(int a=10, int b=20)            // valid, all can be default
```

Default arguments at function proto-type

We know that if a function call appears before the function body, we need to provide the prototype to avoid a compile-time error. Default argument values can be given in the function prototype.
Let us see the following example

```
void show( int a ,  int b=0 );    // function proto-type with default arguments
void main()
{   show( 100 , 200 );
    show( 100 );              //  here default argument 0  is assigned  to  'b'
}
void show(int a, int b)       // don't repeat the default arguments here, already we have given at proto-type.
{   printf("\n%d  %d" , a , b);
}
```

output: 100 200
100 0

remember: Default arguments should ***not be given*** in both places (at proto-type & at function–body).

Ambiguity default arguments with function overloading

```
void show( int a )
{   printf("\n%d " , a );
}
void show( int a, int b=50 )
{   printf("\n%d  %d" , a , b);
}
void main()
{   show( 10 , 20 );     //  no error in this call
    show( 10 );          //   this call makes an ambiguity error
}
```

The function call statement show(10) results in an ambiguity error because this call matches both function definitions. As a result, the compiler is unable to determine which function should be called, leading to an ambiguity error. We need to resolve this issue manually.

Inline functions

For example, you want to have a simple breakfast in a faraway hotel. In this case, going to the hotel, waiting for the order, and returning from the hotel take much more time than preparing and having it at home. This travelling time seems to be an overhead for a simple breakfast. Overhead means the time spent traveling is a greater burden than the time spent preparing at home.

In the same way, when a function is called, control transfers from the calling function to the called function, where it creates parameters and local variables, and after, while returning from the function, these variables will be destroyed. For simple functions having **one or two lines of code**, this task of transferring control between functions and creating-destroying variables seems to be an overhead. Here, this transferring task dominates the execution of code in the function body.

For this problem, the solution is an inline function. An inline function is a simple function having one or two lines of limited code in its body. It is just like a macro function in C. To make any function inline, we need to add the keyword 'inline' before the function name. For example:

```
inline  int  add( int a , int b )
{    return a+b;
}
```

If any function is inline, the compiler substitutes the function's code at every call statement that appears in the program. That is, it replaces all function-call statements with the function's code. Let us see

```
inline  int  findSum(  int  x ,  int  y )
{       return x+y;
}

int  main()
{       int k , A=7,B=8;
        k = findSum( 10 , 20 );
        printf("output  is %d",  k );
        k = findSum( A , B );
        printf("output  is %d",  k );
}
```

→

After compilation, the code can be imagined as

```
int  main()
{       int k , A=7,B=8;
        k = 10 + 20 ;
        printf("output  is %d",  k );
        k = A + B ;
        printf("output  is %d",  k );
}
```

observe that, function-call statement and function-body eliminated.

So, inline eliminates the overhead of function calls, function returns, and the creation and destruction of local variables and parameters. The main purpose of inline functions is to increase the performance of program execution by eliminating function calls.

In c++, member function's body can be written inside and outside of class-body. If function body presented inside class-body, then compiler automatically takes them as inline, so here the keyword inline is optional for all functions which appears inside class body. Remember, 'the code must be in one or two lines'. If more lines of code presented then compiler takes them as normal function. Thus the keyword inline is not a command, just a request to the compiler to take function as in-line. Let us have one example,

```
class  Sample
{       int add( int  a , int b )  // as function body written inside class body, compiler automatically takes as inline function
        {       return a+b;
        }

        int findBig( int a,int b)  // The compiler tries to take it as an inline function  but since the body has too many lines,
        {       if( a>b)                                                    //  it is treated as a normal function instead.
                        return  a;
                else
                        return  b;
        }
};
```

1) If function body is presented outside of class, we need add 'inline' keyword explicitly before the function name. In this case, the body must also be limited to one or two lines of code; otherwise, the compiler will treat it as a normal member function, even if it is declared as inline. Let us see following example, where function body has written outside

```
class  Sample
{     int  add( int  a , int b);
      int  findBig( int a, int b);
};

inline int Sample::add( int a , int b)        // we need to add inline keyword explicitly as body written outside class.
{     return a+b;                             //  otherwise takes as normal function.
}

inline  int  Sample::findBig( int a , int b)  // though declared as inline, but compiler takes as normal member  fn.
{     if( a>b)                                //  because, the code having more lines
         return  a;
      else  return  b;
}
```

Remember, the inline feature can be applied to global functions as well as class member functions.
Suggestion: Just add inline for all functions with code consisting of one or two lines. Don't worry about whether the function body is inside or outside the class block.
Actually, this feature could have been managed by the compiler rather than being provided explicitly in C++.

Friend function and Friend class

If you introduce a person as a friend to your parents, they allow him to come inside your home, where he can then access all your belongings. Similarly, a friend function is non-member of a class, can access all private data of that class. Here, the compiler acts as the parent.

A friend function is a non-member function however it can access all private data of a class.
(Non-member function means: global scope C function and can be called directly without any object)

Sometimes, almost all private members of a class need to be accessed by a non-member function, and in such cases, a friend function becomes the solution. When a function is declared as a friend of a class, it gains access to all the private members of that class. One might wonder why we don't use set() and get() functions instead to access private data. The reason is that a friend function can directly access all private members, making it more efficient and suitable for certain scenarios.

```
class Test
{       private:  int  K;
};
int  main()
{       Test   ob;
        ob.K=100;       // private not accessible
}
```

The main() function is a non-member of the class, so it can't access the private variable K. To make the main() function a friend of the class, we need to declare the function prototype inside the class along with the keyword 'friend'. This is as given below,

```
class Test
{       private:  int  K;
        friend  int  main();    // This declaration tells that main()  is a friend to the class, so compiler allows it to access private
};
int  main()
{       Test   ob;
        ob.K=100;       // no error, now accessible
}
```

Here, the class Test is telling the compiler that main() is its friend, so the compiler allows it to access the private data. The prototype of the friend function can be declared anywhere in the class, either in private or public, and it does not matter. Let's look at one more example.

```
#include<stdio.h>
class Test
{       private:  int  X , Y;
        public:   Test()
                  {   X=100;
                      Y=200;
                  }
                  friend  void  show(Test);    // now show() function becomes friend of class
};
void show( Test   ob )
{       printf(" %d  %d " ,  ob.X , ob.Y );      // private ob.X and ob.Y is accessible.
}
int  main()
{       Test   ob;
        show(ob);
}
```

Friend class

Like a friend function, we can make one class a friend of another class, so that it can access all private members of that class. If class-A is a friend to class-B, then class-B can access all private members of class-A. Let us see the following example.

```
#include <stdio.h>
class  B;                  // this is forward declaration of class-B ,  like function-proto-type;

class A
{       private:  int  x , y;
                  friend class B;
};

class  B
{       public:
        void show()
        {       A   p;
                p.x=10;        // now class-B can access private members of class-A
                p.y=20;
                printf(" %d  %d "  , p.x , p.y ) ;
        }
};
int   main()
{       B  ob;
        ob.show();
}
```

Here, class-A is telling the compiler that class-B is its friend, so the compiler allows class-B to access class-A's private members. These types of applications are used very rarely.

C++ reference operator(&)

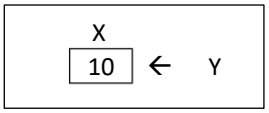
Using C++ reference operator, we can create new name to the existing variable, and this new name works as alias name (nick-name) to the original variable. The original name and reference name access the same memory**. This reference operator was designed to simplify the call-by-reference mechanism in C++.**

In case of C language call by reference, we may need to use '*' operator many times in the called function. This is somewhat complex and tedious due to the repeated use of the '*' operator. If you're a senior programmer, you might have experienced difficulties with the '*' operator. To eliminate this complex use of '*' operator and make the process more straightforward and less error-prone, the reference operator was introduced in C++.

A reference variable is created using the '&' symbol, and this symbol is also used to get address of a variable. However we don't get any confusion because this symbol is used only once while creating reference variable. Let us see how to create reference variable.

syntax: data-type &reference-variable-name = variable-name;

example:
```
int main()
{       int  X=10;
        int  &Y=X;  // reference must be initialized while declaration, otherwise it is an error.
        X++;
        printf("%d  %d ",  X , Y );       //  11    11
        Y++;
        printf("%d  %d ",  X , Y);        //  12    12
}
```



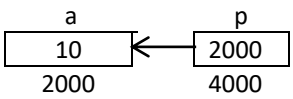
now Y becomes reference variable of X , and these two variables accesses the same memory.

Let us see difference between C pointer and C++ reference.

Pointer usage in C

```
int main()
{   int  a = 100;
    int  *p = &a;
    printf("%d  %d" , a , *p );
    *p=200;
    printf("%d  %d" , a , *p );
}
```

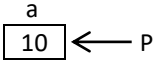
output: 100 100
output: 200 200



reference usage in C++

```
int main()
{   int a = 100;
    int  &p = a;
    printf("%d  %d" , a , p );   // observe, symbol '*' is not used
    p=200;                       //  this reflects in 'a'
    printf("%d  %d" , a , p );
}
```

output: 100 100
output: 200 200



When the original variable is available within a function, creating a reference variable is unnecessary. This example is for demonstration purposes only. The following example illustrates how the reference operator is used in call-by-reference.

References are mainly used in call-by-reference , let us see following example

Call by reference in C	Call by reference in C++

int main()
{ int a=100, b=200;
swap(&a , &b);
printf("%d %d", a , b);
}
 	
void swap(int *p, int *q) // int *p=&a , int *q=&b
{ int t;
t=*p;
*p=*q; // observe, usage of '*' many times
*q=t;
}
output: 200 100
 	
when swap() fn is called, the arguments(&a,&b) are assigned to parameters (p,q) as: int *p=&a , int *q=&b;
	int main()
	{ int a=100, b=200;
	swap(a , b);
	printf("%d %d", a , b);
	}
	
	void swap(int &p, int &q) // int &p=a , int &q=b;
	{ int t;
	t=p; // observe here '*' not used
	p=q; // bcause p&q reference same memory of a,b
	q=t; // this is simple and fast
	}
	output: 200 100
	
	when swap() fn is called, the arguments (a,b) are assigned to parameters (p,q) as: int &p=a , int &q=b;

References in call-by-value (passing reference to a function)

if passing argument to a function is a large object, creating parameter copy at the called function is unnecessary. In this case, to avoid creating a copy of such large arguments, use reference operator to create reference rather than copy. This approach saves both space & time.

```
class String
{       char a[100];
        ------------
};
void show( String ob2 )     // here 'ob2' is created as copy of ob1;  It is like String ob2=ob1;
{       ------------
}
int main()
{       String ob1("hello");
        show(ob1);
}
```

Here each object ob1 & ob2 takes 100 bytes memory. Here parameter ob2 is created as a copy of ob1 at show() function. But it is un-necessary creating ob2 as one more copy for ob1. To avoid this, take ob2 as a reference. This is as given below

```
void show( String &ob2 )       // here 'ob2' is a reference variable. This is like: String &ob2=ob1;
{       ------------
}
```

Returning reference from a function

Returning a reference is also possible, but it should not be a function's local variable. That is, we can't return a reference to a local variable from a function because all local variables are automatically deleted by the compiler when the control returns from the function. So, we can't create a reference to a non-existing variable. The following program shows a compile-time error.

```
int&  test()
{   int  a=100;
    return(&a);     //  fatal error. 'a' will be deleted while returning control , so we can't return reference of local variable.
}

void   main()
{   printf("%d " , test()  );
}
```

This example returns the global variable's reference.

```
int  a , b;

int&  getBig()
{       if( a>b)
            return a;                 // reference is returning to main() function
        else
            return b;
}

int main()
{       printf("enter two values:");
        scanf("%d%d", &a, &b);
        getBig()=1000;              // the big value modified to 1000
        printf("%d  %d", a, b);
}
```

Similarly, returning a reference can be advantageous when dealing with large objects, as it prevents unnecessary creation of temporary object. Let us see following code

```
int  add( int a, int b )
{    return a+b;
}
```

Here before returning a+b value to main() function, the compiler stores the a+b value into a temporary variable. Actually, temporary is a nameless variable created & destroyed by the compiler. For a while, let us say its name is 'temp'. The data type of 'temp' is depends upon a+b value-type. The above function can be imagined with 'temp' as:

```
int  add( int a, int b )
{       int temp=a+b;
        return temp;
}
```

Let this 'temp' is String object like above said programs, then it takes 100 bytes space and also takes extra time to copy the string into temp. Let us see following example,

```
class  String
{       char  a[100];
        ----------
        String  change( char *p)
        {       strcpy(a , p);
                return *this;
        }
};
int   main()
{       String   ob1("hello"), ob2;
        ob2=ob1.change("world");    // the 'temp' will be destroyed after copying into ob2. ( ob2=temp)
        --------
}
```

String temp = *this;
return temp;

Here 'temp' is created as an object of type 'String' class, and it consumes 100 bytes space. It is unnecessary creating such big object while returning. In this case, it is better to return reference like given below

```
String&    change ( char *p )
{       strcpy(a , p);
        return *this;
}
```

Returning reference of *this

More about reference variable

A reference variable must be initialized at the time of its declaration, and once initialized, cannot be changed to refer to different variable.

Reference variable implicitly a pointer managed by compiler. It is an implicit hidden pointer and we don't have to use '*' operator while accessing the original variable.

Some people may mistakenly believe that C++ references can fully replace C pointers. This is wrong. Pointers are used to access data through memory addresses and have a wide range of applications. References, on the other hand, are primarily used in call-by-reference, call-by-value, and return-by-reference scenarios. That is, while passing & returning data between calling and called functions.

---

conclusion

1) In case of call-by-reference, the C++ reference operator is the best solution.
2) In case of call-by-value for large size objects, the reference is the best solution.
3) In case of returning large size objects, the reference is the best to avoid creation of temporary object.
4) in case of call-by-value for small size objects, the copy is best for passing & returning between functions.

Constructor & Destructor

We have already learned how to assign values to an object using the set() function. Now, we will learn how to initialize values to an object through constructor. First, let us recall how we used to perform initialization and assignment for structures and arrays in the C language.

Initialization	Assignment

Assigning values at the time of declaration is said to be initialization. Following code gives an idea how arrays and structures are initialized in C.
 	
int a[3] = { 10, 20, 30 };
 	
struct Date dt = { 10, 5, 2024 };
	Assigning values to each member separately after declaration of variables is said to be assignment.
	int a[3];
	a[0]=10;
	a[1]=20;
	a[2]=30
	
	struct Date dt;
	dt.day=10;
	dt.month=5;
	dt.year=2024; // this is what we practiced in C

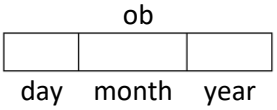
The following code snippet is an example of invalid initialization for class objects

```
class  Date
{   int   day , month , year;
    --------
};
int main()
{   Date  ob={10,5,2010};      //  this type of initialization is not allowed in C++, shows an error at compile time.
    -------
}
```

Many people think that C-style structure initialization may also be supported for C++ objects, but this is incorrect. This results an error. In C++, objects cannot be initialized like in C, according to OOP principles, all operations on object members—such as reading values, writing values, and calculating values—must be performed inside the class through member functions. No operations should be performed outside the class. Thus, we **should not** assign or initialize values to objects in the main() function or any other non-member function. The following examples show some valid and invalid assignments/initializations.

Following is also an invalid initialization

```
class  Date
{   int  day=10 , month=5; year=2010;      // here space will not be created for day, month, year.
    --------                               // so we cannot initialize variables at this place.
    ---------                              // Remember, space creates in object 'ob' at main() fn.
};
int main()
{       Date  ob;    // here space creates for day, month, year in 'ob' ;
        -------
}
```



We can't initialize data members like above shown, because space for data members will not be created inside the class. (Remember, member's space creates in the object like above pic). Therefore, it shows an error. The following code also explains invalid assignments to the objects.

example3: invalid assignment

```
class  Date
{   public:  int   day , month , year;
    --------
    --------
};
int main()
{   Date  ob;
    ob.day=10;
    ob.month=5;
    ob.year=2010;
    -------
}
```

Even though this code is correct, we should not assign values in this way. This is not a good practice style in oop. We should assign through set() function. This is given in right side block. This concept is already discussed before. That is, everything must be done inside class block.

Following is valid assignment

```
class  Date
{   private:  int   day , month , year;
    pulic:
          void set(int day, int month, int year)
          {   this->day=day;
              this->month=month;
              this->year=year;
          }
          --------
          --------
};

int main()
{   Date  ob;
    ob.set( 10 , 5, 2010);
    -------
}
```

This is right way of assigning values to object.

The set() function is used to assign the values to object members, whereas constructor is used to initialize the values to object members. Let us move to the constructors.

Constructor

A constructor is a special member function that has the same name as the class and is automatically called when an object is created. It is used to initialize the object's members with values. A constructor does not have a return type, not even void. Let us see following example

```
class   Test
{       ----
        Test()          // this is constructor function, and automatically called when space is created for object .
        {       ----
                ----
        }
        ----
}
```

Let us see one example program.

```
class  Finder
{       int  sum=0 , product=1;
        ------
        ------
};
```

We cannot assign values to data members like sum = 0 or product = 1 as shown above, since memory is not allocated for sum and product at that point.

```
class  Finder
{       int sum , product;
        Finder()      // this is constructor
        {       sum=0;
                product=1;
        }
        ------
}
```

This is the correct code how to initialize ‘sum’ and ‘product’ through constructor.

The creation of memory space for object is nothing but creation of memory space for each data member in the object. When memory is allocated for data members, there are some initial tasks may be involved, such as initializing values to data members like sum=0, product=1, or allocating resources like creating dynamic memory for pointers, opening files, connecting to networks, connecting to printers, etc. In this way, we need to initialize the values as well as allocate resources which are required by the object when space is created. These tasks are typically written inside the constructor to make the object ready for use.

A constructor is a special member function that has the same name as the class and is automatically called when an object is created. It is used to initialize the object's members with values. A constructor does not have a return type, not even void. Constructors can be overloaded too. A constructor that takes no arguments is called a default constructor, while one that takes arguments is called a parameterized constructor. Constructors are like any other function and can contain any code. This makes the object ready for use. Let us see the syntax

```
syntax:  class  class-name
         {      ----
                ----
                class-name()                        // this is default constructor
                {      ----
                       ----
                }
                class-name ( parameters list 1)     // this is parameterized constructor1
                {      ----
                       ----
                }
                class-name ( parameters list 2)     // this is parameterized constructor2
                {      ----
                       ----
                }
                ----
         };
```

This example explains how constructors will be called in the program.

```
#include<stdio.h>
int main()
{
   Date  ob1;

   Date  ob2(2022);

   Date  ob3(10,5,2022);

   ob1.show();
   ob2.show();
   ob3.show();
}
```

```
Date is 00-00-0000
Date is 01-01-2022
Date is 10-05-2022
```

```
class Date
{   int d,m,y;
    public:
    Date()              // this is default constructor
    {  d=m=y=0;
    }

    Date(int y)         // this is parameterized constructor1
    {  d=m=1;
       this->y=y;
    }

    Date(int d, int m, int y)  //parameterized constructor2
    {  this->d=d;
       this->m=m;
       this->y=y;
    }

    void show()
    {  printf("\nDate is %2d-%2d-%4d",d, m, y );
    }
};
```

The instructions in the main() function are converted as given below
The instruction 'Date ob1'' is converted into two instructions by the compiler as,

```
Date  ob1;      // creates memory space for ob1
ob1.Date();     // calling default constructor to set 0 to d,m,y.
```

Similarly, the instruction 'Date ob2(2022)' is converted into two instruction by the compiler as

```
Date  ob2;
ob2.Date(2022);  // calling parameterized constructor 1
```

The instruction 'Date ob3(10,5,2022)' is converted into two instructions as

```
Date  ob3;
ob3.Date(10,5,2022);  // calling parameterized constructor 2
```

Constructors and the set() function may seem similar in some cases (since both assign values to members), but they serve different purposes. A constructor is used to initialize values to an object at the time of its creation, whereas the set() function is used to assign or update values later in the program, after the object has been created. Like the set() function, a constructor can also validate data before initialization.
So the constructor prepares the object ready for use. While destructor is quite opposite to constructor, where it releases (deletes) such allocated resources before destroying the object

In a typical scenario, a class is written by a person 'X', while the main() is written by another person 'Y'. 'X' anticipates all the possible functionalities 'Y' might need later. Therefore, it's better to write all possible constructors in the class. Following program is one more example on Time class.

```
int main()
{
        Time ob1;

        Time ob2( 7450 );

        Time ob3(4,20,40);

        ob1.show();
        ob2.show();
        ob3.show();
}

output is: 0  0  0
output is: 2  4  10
output is: 4  20  40
```

```
#include<iostream>
using namespace std;
class Time
{       int h,m,s;
        public:

        Time()
        {   h=m=s=0;
        }

        Time(  int n )
        {   h=n/3600;
            m=n%3600/60;
            s=n%3600%60;
        }

        Time (int h, int m, int s)
        {   this->h=h;
            this->m=m;
            this->s=s;
        }

        int  show()
        {    printf("\n %d:%d:%d", h, m, s);
        }
};
```

We know objects can be declared not only in the main() function but also anywhere in the program. However, constructors are called for every object creation, regardless of where the object is declared. Let us see following program

```
int  main()
{
    Test  ob1;
    Test  ob2;
     check();
}
```

Here the constructor automatically called 3 times for ob1, ob2, ob3 while creating space.
output: hello hello hello

```
#include<stdio.h>
class  Test
{
    Test()
    {
         printf("hello");
    }
    void  check()
    {    Test ob3;
         -----
    }
};
```

If no constructor is written in the class, then compiler automatically creates a default constructor with an empty body. For example Test() { }

```
class  Test
{       int  a , b;
        ---------
        ---------
};
int    main()
{       Test  ob;
        ----------
        ---------
}
```

Observe how the default constructor added by the compiler

```
class  Test
{       int  a , b;

        Test()          // this constructor will be created by compiler
        {
        }
};
int    main()
{       Test  ob;
        ------
}
```

This program explains how C++ code converts into C code by the compiler with the help of 'this' pointer.

C++ code with two constructors

```
class  Test
{   int  a, b;

    Test()
    {   a=b=0;
    }

    Test( int a, int b )
    {   this->a=a;
        this->b=b;
    }
};

int   main()
{
    Test  ob1;

    Test  ob2(10,20);

    ------
    ------
}
```

Code after converting to C

```
struct  Test
{    int a,b;
};

Test( struct Test *this )
 {      this->a = this->b = 0;
 }

Test ( struct Test *this , int a, int b )
 {   this->a=a;
     this->b=b;
 }

 int    main()
 {
     struct  Test  ob1;
     ob1.Test();          //   this line added by the compiler to call
                             default-constructor

     struct  Test  ob2;
     ob2.Test(10,20);  // this line too added by the compiler
     ------                // this calls parameterized constructor
     ------
 }
```

finding radius of circle

```
int  main()
{   Circle  ob1;
    ob1 . find( );
    ob1 . print( );

    Circle  ob2(7);
    ob2 . find( );
    ob2 . print( );
}
```

Above main() fn can also be written as

```
 int  main()
{   Circle  ob;
    ob . find( );
    ob . print( );

    ob. set(7);
    ob . find( );
    ob . print( );
}
```

op: area is 78.5
perimeter is 31.4
area is 153.86
perimeter is 43.96

```
class  Circle
{   float radius, area, perimeter;
    Circle()
    {   radius=5;      // just for demo '5' is taken
    }

    Circle( int r )
    {   radius=r;
    }

    void set( int r )
    {   radius=r;
    }

    void  find()
    {   area = 3.14 * radius * radius;
        perimeter = 2* 3.14 * radius;
    }

    void  print()
    {   printf("area is %.2f",  area);
        printf("perimeter is %.2f", perimeter);
    }
};
```

Inserting values in list and printing on the screen, here constructor creates array of user wanted size.

```
int   main()
{   List  ob(10);      // list size (array-size)
    ob.add(14);
    ob.add(5);
    ob.add(11);
    ob.add(16);
    ob.show();
}
```

output: 14 5 11 16

```
class  List
{   int  *arr, size, count;
    public:
    List( int size )
    {   arr=new int [ size ];
        this->size=size;
        count=0;
    }
    void   add( int x )
    {   if ( count==size )
        {       printf("error , array full ");
                return;
        }
        arr [ count++ ] = x;
    }
    void  show()
    {   for ( int i=0; i<count; i++ )
            printf("%d  ", arr[i] ):
    }
};
```

Destructor

this is opposite to a constructor. A destructor is automatically called just before an object is going to be destroyed. Its name is the same as the class name, but prefixed with a tilde (~) symbol. Unlike constructors, destructors cannot be overloaded, so only one copy is allowed in a class. Destructors are mainly used to delete dynamically allocated memory, close files, and release other resources.

```
syntax:     ~class-name()     // this is destructor
            {     ------
                  ------
            }

example:    class  Test
            {     public:  Test()
                  {     printf("\n  I am at constructor ");
                  }
                  ~Test()
                  {     printf("\n  I am at destructor");
                  }
            };
            int   main()
            {     printf(" \n before ob1");
                  Test  ob1;          // here constructor will be called for ob1
                  printf(" \n before ob2");
                  {     Test  ob2;    // here constructor will be called for ob2
                  }                   // here destructor will be called for ob2 before going out of this block
                  printf(" \n main end ");

            }  // here destructor will be called for ob1
```

```
op:  before ob1
     I am at constructor   (for ob1)
     before ob2
     I am at constructor   (for ob2)
     I am at destructor    (for ob2)
     main end
     I am at destructor    (for ob1)
```

---

creating & destroying dynamic memory in constructor and destructor. (read DMA chapter before this)

```
class  List
{     int   *arr;
      List()
      {     arr=new  int[10];
      }
      ~List()
      {     delete arr;
      }
};
```

Demo how to **delete previously allocated memory** when setting a new string to an object.

```
int main()
{   String ob("hello1");
    ob.print();

    ob.set("hello2");
    ob.print();

    ob.set("hello3");
    ob.print();
}
```

The set() function is called many times to update the string in ob. Before updating to a new string, the old string should be deleted. For example

```
if( p!=NULL)
    delete p;
```

```
#include<stdio.h>
#include<string.h>
class String
{   char *p;
    public:
    String() { p=NULL; }
    String(char *p)
    {   this->p=new char[100];
        strcpy(this->p , p );
    }
    void set(char *p)
    {   if( p!=NULL)
            delete p;  // deleting previous allocated memory
        this->p = new char[100];
        strcpy( this->p , p );
    }
    void print()
    {   printf("\nstring is:%s",p);
    }
};
```

Expect output : how many times the constructor and destructor will be called

```
int main()
{   int i;
    for( i=0; i<10; i++)
    {   String ob;
    }
}
```

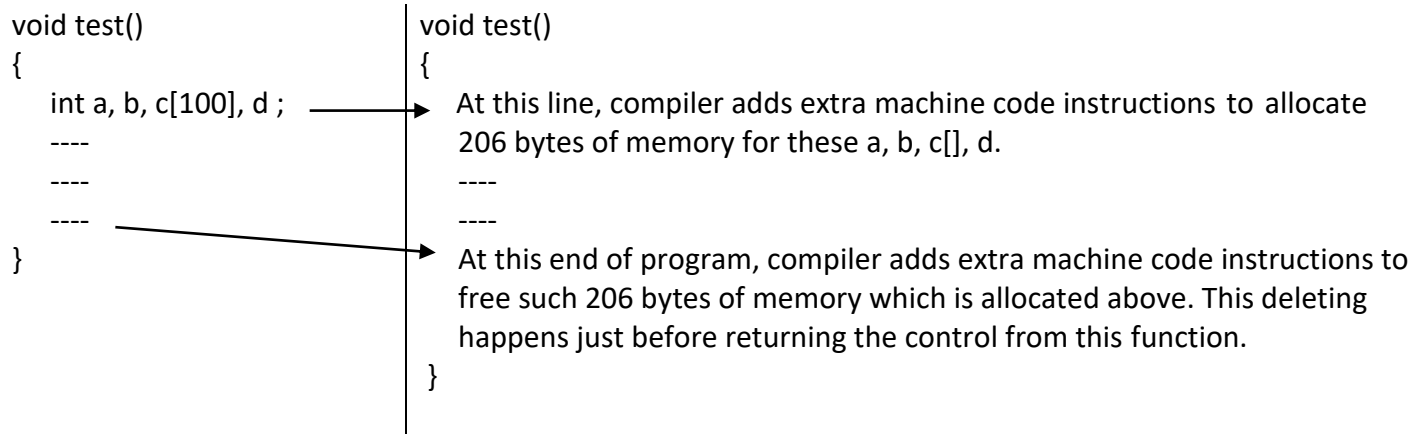
```
class String
{   public:
    String()
    {   printf(“hello”);
    }
    ~String()
    {   printf(“world”);
    }
};
```

Dynamic Memory allocation (DMA)

Memory allocation means creating memory space for variables, arrays and pointers in the RAM during compile-time or run-time.
We have two memory allocation systems ① static allocation ② dynamic allocation
Allocating memory at compile time is said to be static allocation, this is created and managed by the compiler. Of course, from the beginning of this book, we have been using this static allocation system only.
Allocating memory at runtime based on input data size is said to be dynamic allocation.

Static allocation system

if size of data is known at coding time then static allocation is the good choice. The static allocation system is managed by compiler by adding necessary machine code instructions in the program. The compiler adds extra instructions to our program for allocation/de-allocation memory for our variables. So programmer need not to bother about how to create and destroy variable's memory in the program.
Let us see following example



Compiler creates 206 bytes for **int a, b, c[100], d**. It adds extra instructions for allocation & de-allocation of memory as shown above. [It creates memory in stack as stack.push(206) and stack.pop(206)]
The static allocation system is much suitable when we know the size of data like employee-name(30 chars), address(50 chars), pin-code(4bytes for int), phone-number(4bytes for int),…

Dynamic Memory Allocation System in C++

In many real time systems, the size of data is not known or cannot be expected until runtime. Here dynamic system is the choice. Dynamic allocation refers to allocation of memory during execution of program. Here programmer has to write special instructions to allocate & de-allocate memory with the help of pointers.

This DMA system provides us to create, expand, reduce, or even destroy the memory at runtime. This feature is very much useful for efficient memory management especially while handling huge collection of data using 'data structures'. For example arrays, lists, stacks, queues, trees, graphs, etc. To work with this feature, C++ provides two operator keywords "**new and delete**".

Before going to **new & delete** operators, first we will learn difference between operator and a function. Operator is an inbuilt task of compiler whereas function is an outside task from the compiler. Most of the time, operator is a symbol which is associated with some task. For example, the operators are + - * < > = etc. The operation "10+20" is directly supported by compiler and no header file to be included to support this task.

Here compiler generates necessary machine instructions for adding these 10+20 values. Operators are having simple syntax, easy to use, provide compile time error checking, and also no header files to be included. Whereas function is a separate task and need to include header files. Therefore operators are simple to use compared to functions.

The C functions malloc(), calloc(), and free() can also be used in C++ for dynamic memory allocation (DMA), just like in C. However, they have a heavy syntax, so C++ designers alternatively provided 'new' as a replacement for malloc() and 'delete' for free(). Remember, **new** and **delete** are operators, not functions like malloc() and free().

'new' operator: It allocates user wanted size of memory and gives starting byte address of such memory. Unfortunately, if enough memory is not available to allocate then **'new'** returns NULL value.

Syntax: **pointer = new value-type;** // creates space for single item

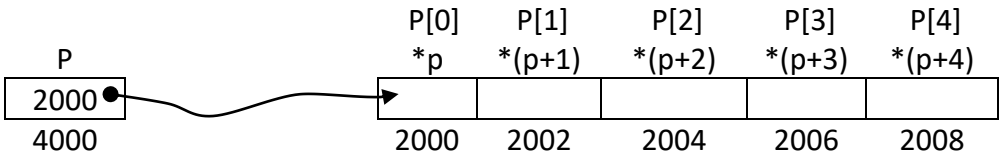
pointer = new value-type[5]; // creates space for 5 items (array of 5 items)

Here value-type can be int , float , char , class , structure , etc.

For example, creating memory for 'n' integers

```
int  *p , N;
printf("enter no.of values:");
scanf("%d",  &N);
p=new  int[N];
```

Let us see, how above allocated memory is mapped to the pointer 'p'



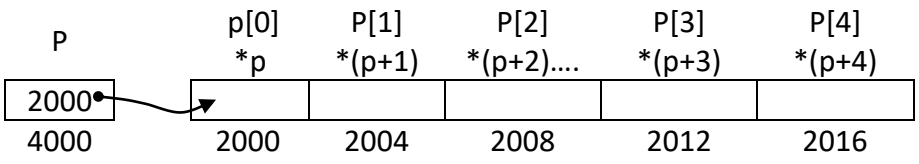
How dynamic memory is accessed?

Accessing this memory is same as accessing array elements through pointer.
Remember, the pointer expression *(p+i) can be written in array style as p[i].
Observe the following expressions

- *(p+0) → p[0] access the first location (first 2bytes , 2000+2001)
- *(p+1) → p[1] access the second location (second 2bytes , 2002+2003)
- *(p+2) → p[2] access the third location
- *(p+i) → p[i] access the(i+1)th location.

If you are creating memory for floating point values, then **float*** type pointer is required.
For example creating memory for **'N'** floats. Here 'N' is the input value

```
float  *p;   int N;
printf("enter no.of values:");
scanf("%d", &N);
p=new  float[N];
```



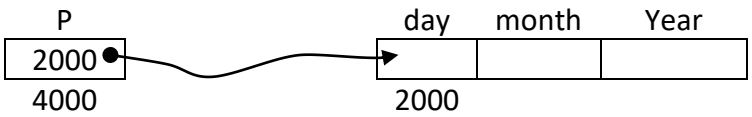
Demo program, accepting 'N' integers and printing sum of all values

```
#include<alloc.h>
int main()
{       int  n, i, *p, sum;
        printf("enter no.of input values:");
        scanf("%d", &n);

        p = new  int[n];
        if(p==NULL)
        {       printf("\n error, memory not created");
                return 0;
        }
        for(i=0; i<n; i++)
        {       printf( "enter any value :");
                scanf( "%d", &p[i] );
        }
        for(i=0; i<n; i++)
        {       sum=sum+p[i];
        }
        printf("sum is : %d  ",  sum );
        return 0;
}
```

In the above, the "new" creates memory for n-integers, later accessed through the pointer **'p',** remember accessing elements using pointer is same as accessing array elements using [ ]. Here the input size can be anything, thus it makes efficient usage of memory.

Creating memory for class objects: we can create class objects dynamically using new operator. When we create class objects using the new & delete then compiler automatically calls the constructor and destructors. Let us see with Date class example.

```
Date *p, *q;
p=new Date;  (or)  p=new Date();      // creating single Date object
q=new Date[3];                        // creating array of 3 Date objects
```



Using 'p' , we can access object members just as in structure in C language.

Using 'q' , we can access array objects just as accessing array elements using pointer in C language.

Let us see following example

Example: Creating Date object in dynamic allocation using new operator.

```
class  Date
{       int d,m,y;
        Date()
        {   d=m=y=0;
        }
        Date( int d,  int m, int y)
        {   this->d=d;   this->m=m;  this->y=y;
        }
        void show()
        {   printf(" Date is : %d - %d - %d" , d , m , y );
        }
};
int main()
{       Date  *p;
        p=new Date();
        p->show();   or  (*p).show();       //  Date is :  0 - 0 – 0

        p=new Date(10,5,2023);              // this calls the parameterized constructor
        p->show();                          //  Date is : 10 - 5 - 2023
}
```

Dynamically creating 3 array objects and also calling 3 parameterized constructors, this is the extension to above program.

```
int main()
{       Date *p;
        p = new Date[3]  {  Date(10, 5, 2020)  ,  Date(4, 5, 2022)  ,  Date(10, 5, 2024)  };
        p[0].show();    or    p->show();
        p[1].show();    or    (p+1)->show();
        p[2].show();    or    (p+2)->show();
        return  0;
}
```

above code can be written using loop as

```
for( i=0; i<4; i++)                          for( i=0; i<4; i++)
{       p[i].show();            (or)         {       p->show();
          (or)                                       p++;          // increments the 'p' by object-size
        (p+i)->show();                       }
}
```

Demo: Comparing two dates whether they are equal or not, creating dynamic memory for objects using new and delete operators.

```
#include<stdio.h>
class  Date
{   int d, m, y;
    Date()
    {   d=m=y=0;
    }
    Date( int d,  int m, int y )
    {   this->d=d;   this->m=m;  this->y=y;
    }
    bool compare( Date *q )
    {   if( d == q->d && m==q->m && y==q->y )
            return true;
        else return false;
    }
};
```

```
int main()
{   Date  *p, *q ;
    p=new Date(10,4,2020); //  creating single object
    q=new Date(10,4,2020); //  creating single object
    if ( p->compare(q)==true)
        printf("equal");
    else
        printf(" not equal ");
}
```

Demo program on strings, here two programs are explained with static and dynamic allocation to keep the string data. The following add() function in the String class concatenates two string objects and returns the result. Initially, we assign two sample strings, 'hello' and 'world' to objects ob1, ob2.
We have used a static array 'char a[100]' to store the input string in first program .

```
#include<stdio.h>
#include<string.h>
class String
{       char  a[100];
        public:
        void set( char str[] )
        {   strcpy(a , str );
        }
        String add( char str[] )
        {   String t;
            strcpy(t.a , a);
            strcat(t.a , str );
            return t;
        }
        String add( String ob2)
        {   String t;
            strcpy(t.a , a );
            strcat(t.a , ob2.a );
            return t;
        }
        void show()
        {   printf("\noutput is:%s", a );
        }
};
```

```
int main()
{       String ob1,ob2,ob3;
        ob1.set("hello");
        ob2.set("world");

        ob3=ob1.add(ob2);
        ob3.show();             //  helloworld

        ob3=ob1.add(" hi ");
        ob3.show();             // hello hi
}
```

This is same as above program with small modification, instead of static array, we have taken dynamic array to store the string data.

```
#include<stdio.h>
#include<string.h>
class String
{     char  *a;
      public:
      void set( char str[] )
      {     a=new char[ strlen(str)+1 ];  // +1 for null-char
            strcpy(a , str );
      }
      String add( char str[] )
      {     String t;
            t.a=new char[ strlen(a) + strlen(str)+1 ];
            strcpy(t.a , a);
            strcat(t.a , str );
            return t;
      }
      String add( String ob2)
      {    String t;
           t.a=new char[ strlen(a)+strlen(ob2.a)+1 ];
           strcpy(t.a , a);
           strcat(t.a , ob2.a );
           return t;
      }
      void show()
      {    printf("\noutput is:%s", a );
      }
};
```

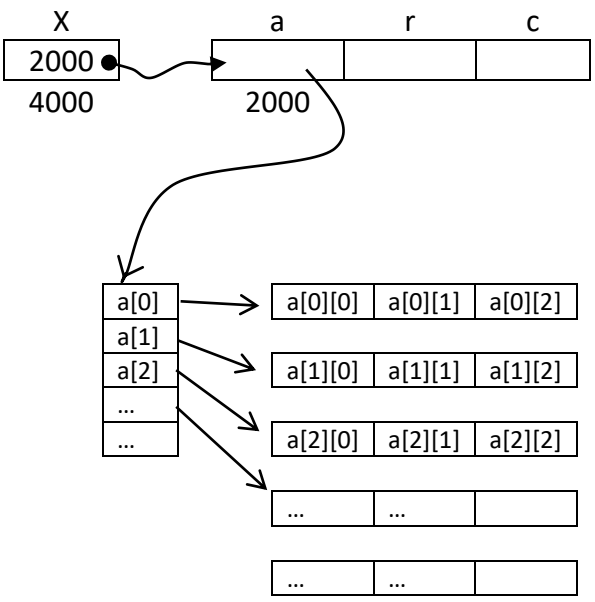
```
int main()
{
        String ob1,ob2,ob3;

        ob1.set("hello");
        ob2.set("world");
        ob3=ob1.add(ob2);
        ob3.show();

        ob3=ob1.add(" hi ");
        ob3.show();
   }
```

Addition of two matrix using DMA. Here we have taken Dynamic 2D array to store matrix data as well as the objects created using DMA

```
#include<stdio.h>
class Matrix
{   int   **a ;                  // pointer 2D array
    int  r , c;
    public:
    Matrix() { r=c=0; }

   Matrix( int r, int c)
   {  int i;
      a=new int* [r];            // creating 'r' rows
      for(i=0; i<r; i++)
           a[i]=new int[c];      // creating 'c' columns
      this->r=r; this->c=c;
   }

   void scan()
   {  int  i , j;
      for(i=0; i<r; i++)
      {  for(j=0; j<c; j++)
         {  printf("enter value of a[%d][%d]:",i,j);
            scanf("%d", &a[i][j]);
         }
      }
   }

   void show()
   {  int i,j;
      for(i=0; i<r; i++)
      {  for(j=0; j<c; j++)
            printf("%4d ",a[i][j] );
         printf("\n");
      }
   }

   Matrix* add( Matrix *y )
   {  int i,j;
      Matrix *t=new Matrix(r,c);
      for(i=0; i<r; i++)
      {  for(j=0; j<c; j++)
            t->a[i][j]=a[i][j]+y->a[i][j];
      }
      return t;
   }
};
```

```
int main()
{  Matrix  *x, *y, *z;
   x = new Matrix(2,3);
   y = new Matrix(2,3);

   printf("enter 1st matrix data\n");
   x->scan();

   printf("enter 2nd matrix data\n");
   y->scan();

   z = x->add(y);
   z->show();
}
```

following pictures shows how object 'X' occupies memory space in the RAM.



Hiding and Abstraction

Hiding

Keeping items in private section is said to be hiding. We have already discussed previously, the class data is operated through its functions. The functions like set(), get(), process(), print(), etc. As we handle class data (objects) through these functions, data members **need not be accessed** directly from outside class. So we keep in private while coding the class. This prevents the direct access from outside class, thus avoiding misuse of data. This also reduces the pollution of variable in the program. Sometimes, some functions also need **not to be** accessed from outside class, such functions also kept in private. So keeping members in private section is said to be hiding.

Abstraction

in simple terms, abstraction refers hiding unnecessary details and showing necessary details.
Let us take library functions pow() and sqrt() in C language, we call these functions without knowing how they calculate result in the program. These functions are hidden from us and are made available through the math.h header file. This is one simple abstraction in the programming.
We can have several levels of abstraction, for example, function1 calls the function2, which in turn calls the function3. This creates a hierarchical level of functional abstraction.

In the software industry, the written classes are kept or hidden in separate files like header files, and they provide documentation and help files on how to use the classes in our program. For example, washing machine manufacturers provide a manual book on how to use it properly. In the same way, they provide information or help files how to use these hidden classes.

In our daily lives, we often use many words in terms of abstraction without even realizing it. Vehicle→two-wheeler→Honda→Activa. The word 'Vehicle' represents higher level abstraction than Activa. Device → electronic device → cell phone → iPhone. The word Device represents higher level abstraction. Medicine tablet name refers one type of abstraction where it abstracts the compositions used.

Abstraction refers to the concept of hiding complex details and presenting only the essential information to the user. It is a way of providing a simplified access interface to the user while hiding the internal working mechanism of a device or system. For example, a power switchboard in our home is used to control fans, tvs, lights, motors, etc. The switchboard is a good user-friendly interface for controlling these devices and also hides their internal working mechanism. The user may or may not know how they work; He simply controls them through switches.

Let's consider another familiar example that many authors use in books to illustrate this concept more clearly. The car brake system is completely abstracted from the driver. The brake system can be disk, drum, or hydraulic. The driver may not be aware of the type of brake system and how it works. They simply press the brake pedal to stop the car. In this case, the brake pedal serves as a user-friendly interface, visible and accessible to the driver, while its underlying functionality remains hidden.

Abstraction can be seen in two ways **data abstraction and functional abstraction.**
A cell phone incorporates various sub-devices like GPS, sound, microphone, and camera. These components exemplify data abstraction (hidden to user), while the buttons on the device represent the functional abstraction.

Let us see one code example on C++

class1 example

```
class  Student
{       int   idno, marks1, marks2;
        char  name[30], address[50];
        int  total, average, rank;
        public:
        void  showAddress()  { … }
        void  showMarks()  { … }
        void  showRank()  { … }
        void  showAll()  { --- }
};
```

class2 example

```
class  Student
{       int   idno, marks1, marks2;
        char  name[30], address[50];
        int  total, average, rank;
        public:
        void  showAll()  { ------- }
};
```

In general, classes are written by a person (X), while the main() function is written by another person (Y). The person Y is the *user*, who uses the classes. Here, X writes different functions to display data in various ways, as shown in the **class1** example. The user, Y, calls these functions based on their requirements.

For instance, if Y wants to display only the rank of a student, they call the showRank() function. This function prints only the rank details, without printing other details like the address or marks. Thus, we can say that the showRank() function prints only the rank and hides other details. This process of writing specific functions to show necessary details while hiding the unnecessary ones is known as **abstraction**.

In **class2**, there is only one function, showAll(), which displays all the details of a student. If the user wants to print only the rank, there is no showRank() function as in the class1 example. The user has no choice but to call the showAll() function, which unnecessarily prints all details. Therefore, class2 provides less abstraction than class1. This example illustrates a basic level of abstraction, but there are different types and levels of abstractions.

In programming also abstraction is implemented in two ways: **data abstraction and functional abstraction.**

```
class  Date
{   private:
    int day, month, year;
    public:
    void setDate(…) { … }
    void setYear(…) { … }
    void  incrementDate() {…}
    void  decrementDate() {…}
    void  showDate() {…}
    void  showYear() {…}
};
```

Let this class developed by a person 'X'

```
int  main()
{   Date   ob;
    ob.setDate(10, 4, 2020 );
    ob.incrementDate();
    ob.showDate();
}
```

Let this main() fn developed by a person 'Y'

The functions such as setDate(), setYear(), setMonth(), showDate(), showYear() comes under data abstraction. These are used to read/write values of data members from outside of class. These acts as control interface to data members with validations and other activities.

The functions such as incrementDate(), decrementDate(), compareDates() comes under functional abstraction. These are used to compute the data for finding certain result of user requirements.

From the first example, the showRank() fn shows only the rank and hides other details such as address, marks, and total (data abstraction). The showRank() fn also hides the logic how it calculates the rank from marks, which is called functional abstraction. Thus, the showRank() fn serves as both data abstraction and functional abstraction.

In this way, writing suitable controlled interface functions to the class makes good abstraction.
In programming, the abstraction is done through private, public, functions, classes, and namespace.
Using these tools we can make several different levels of abstractions.
The class abstracts the data and its functions, private hides the internal things. The function overloading, operator overloading, function overriding is one type of abstraction. The concept of inheritance itself represents hierarchical abstraction, where one class inherit the data and functions from other classes. This is one type of abstraction.

In C++, virtual functions are one type abstraction. For instance, Vehicle *p = new Vehicle (new Car()); we'll explore this virtual functions concept at end of this book. **The primary goal of abstraction is to reduce dependencies and complexities, making systems easier to use and maintain.**

Understanding complete abstraction can be challenging for beginners. A class itself is a form of abstraction, and writing suitable functions to process class data is another type of abstraction. Similarly, concepts like inheritance, function overloading, operator overloading, function overriding, and virtual functions also contribute to abstraction at various levels. All these features promote abstraction in different ways throughout a program

OOP (Object Oriented Programming)

The evolution of programming languages is a constant process**.** Initially, machine language was developed, followed by assembly languages, high-level languages, structured high-level languages, object-oriented languages, and script languages. Examples include 8085, 8086, Fortran, Pascal, C, C++, Java, Python, and more. As software requirements expand, new languages are often invented to address evolving needs. These new languages often introduce innovative concepts and approaches.

C is a structured and function-oriented language. Structured means the language provides good control statements along with building blocks of code defined by braces { } . And also breaks down a big program into several functions. Therefore, it is said to be a structured programming language as well as a function-oriented language.

Before the C language, older languages like BASIC, Fortran, Algol, and Pascal did not provide proper control statements. These were considered unstructured languages. In contrast, C offers robust control statements with building blocks using braces {}, making it a well-structured language.

In C, large programs are divided into several modules at the design level, and these modules are implemented as functions during coding. Ultimately, the entire program becomes a collection of functions. Therefore, C is often referred to as a function or procedure oriented language. In Pascal and certain other languages, functions are called **procedures** or **routines**.

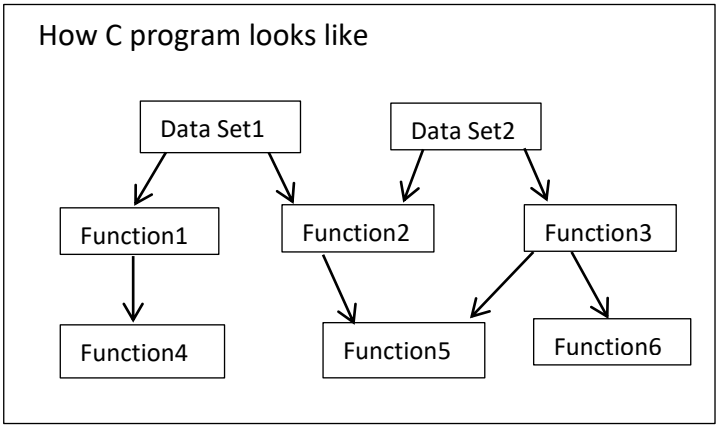
The main issue with function-oriented languages is that all functions are defined globally, meaning any function can be called from any part of the program. As the number of functions increases in larger projects, managing them becomes increasingly complex. Once the number of functions exceeds 100, it becomes difficult to remember and organize them effectively. In fact, the C language tends to struggle when the program size exceeds 50,000 lines of code.

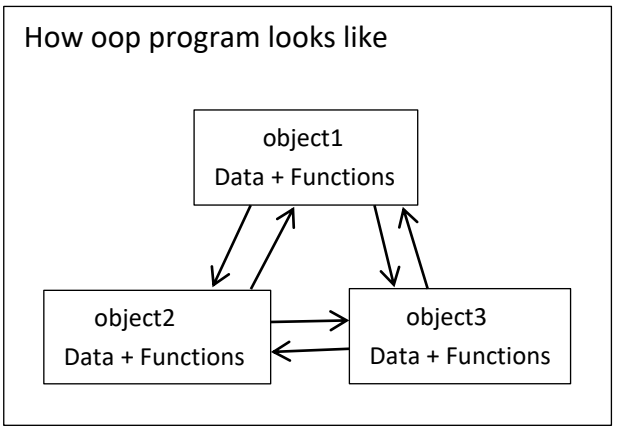
We know that a program is a collection of data and code. In large programs, data is typically represented using structures, and code using functions. In C, both structures and functions are defined in the global scope. The problem arises when the number of structures and functions increases—it becomes difficult to determine which structure is being processed by which function. Everything gets mixed up, leading to confusion for the programmer.

Object-oriented programming is a way to break a big program into smaller parts called objects. Each object has its own data and functions. We write instructions using these objects, like object.scan(), object.find(), or object.print(). At first, this idea was called object-based programming, but later it became known as object-oriented programming. Later, some rules and regulations were added to this programming approach.

Finally object oriented programming is a way of writing programs by following some principles and approaches. We have mainly three standard principles: **encapsulation, polymorphism and inheritance.**
By following these principles one can solve complex programs easily. No doubt these approaches were designed and tested thousands of times on different projects before introducing to the world.

Structured languages mainly focus on functionality, whereas object-oriented languages focus on data alongside functionality. In structured languages, programs appear to be collections of functions, whereas in object-oriented programs appear to be data with functions (classes).





If there are no ideal structural approaches to force the programmer to write the code in systematic way then everybody writes one's own style and makes the program complex and un-readable to others. For example, if there are no predefined traffic rules in city roads, everyone travels as per one's convenience and makes traffic disorders and cause inconvenience to others; so the traffic structures such as signals, dividers, zebra lines and indicators force the traveller to follow rules in order not to cause confusion.

In structured languages, there were no good ideal approaches to write big programs in systematic way. Here everyone writes one's style and causes complexity to others. In object oriented languages everyone writes same style of coding by following these principles and approaches. So we get uniform style of coding thereby easy to understand & update. When we follow uniform style coding, others can follow easily.
The main three approaches are: encapsulation, polymorphism, inheritance.

Encapsulation

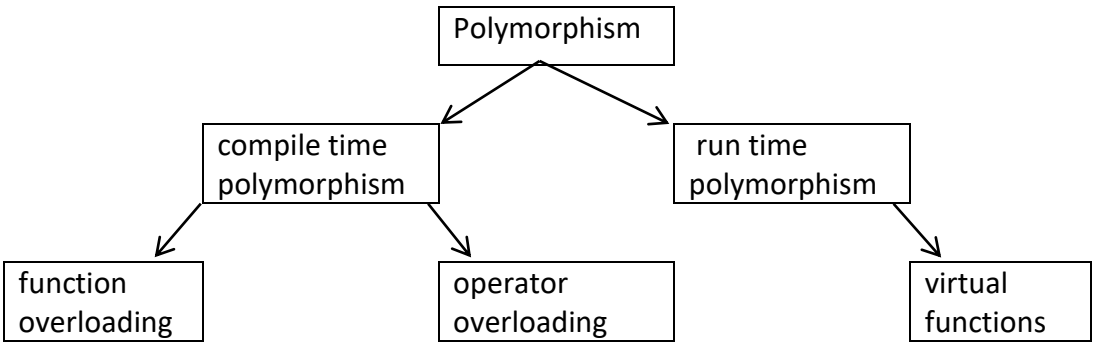
Encapsulation refers to the concept of packing data items and its manipulation functions together as single unit. The exact meaning of encapsulation in OOP includes: **packing, hiding, abstraction, and modularity**.
In C++, a **class** is a tool that supports encapsulation, offering packing, hiding, abstraction, and modularity.
In contrast, in the C language, a **structure** groups several data items into a single unit, supporting only packing but not the other features.

We have already seen that a class is a tool that combines data and its manipulation functions into a single unit. It encapsulates or wraps the data and functions together, promoting data hiding, abstraction, and modularity. Data hiding and abstraction can be seen as by-products of using a class. We have already discussed classes and their syntax. Additionally, namespaces contribute to encapsulation by preventing name collisions between classes, variables, and functions.

Polymorphism

The term 'polymorphism' is derived from Greek, where 'poly' means many and 'morph' means form.
'Poly' refers to many similar related things or actions, while 'morph' refers to transforming them into a single thing or action.

In C++, polymorphism refers to having many forms with the same interface. Let's look at a real-world example. Different cars use different braking systems like drum, disk, hydraulic, air, ABS, dual, etc. However, all cars have the same brake pedal interface. In other words, pressing the brake pedal is the same in all cars, and drivers may not even be aware of which braking system is in use. Similarly, in C++, polymorphism allows objects, functions, or operators to behave differently depending on the context in which they are used.
Polymorphism is classified as shown below.



In the C language, we cannot assign the same name to more than one function; each function carries unique name. However, in C++, the same name can be given to two or more functions, provided their parameters are different. This feature falls under the concept of compile-time polymorphism in OOP.

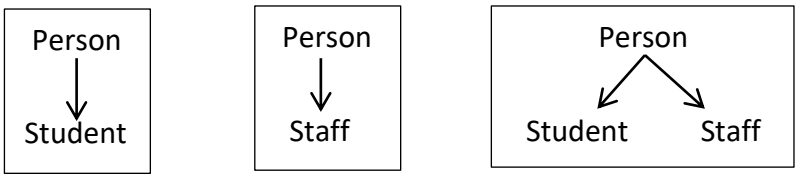
This feature makes it more convenient to remember function names related to the same action, and it also makes the functions easier to understand and use. Defining multiple functions with the same name but different parameters is known as function overloading, which is a concept of compile-time polymorphism in C++. We have already discussed this in previous topics.

Operator overloading is a shorthand term for operator function overloading, and it also falls under static binding or early binding. In this case, instead of using a function name, we use the operator symbol when writing the function body. We will see later.

Compile-time polymorphism is also called static binding or early binding because the function-call statements are linked or bound to the function body during compilation. In contrast, in runtime binding, function-call statements are linked or bound to the function body during program execution, based on input values. This concept is also known as late binding, dynamic binding, or runtime binding. In this case, the functions are called virtual functions. Nowadays, this virtual functions concept is also function-over-riding (this topic will be covered in the next chapters).

Inheritance

Inheritance is the concept of sharing one class's properties with another class, similar to how a child inherits properties from a parent, such as a car, bike, home, etc.
It is a fundamental concept in object-oriented programming (OOP) that allows a class (derived class) to inherit data properties and behaviours (methods) from another class (base class). This promotes code reusability, flexibility and establishes a hierarchical relationship between classes. Let us take college administration example



```
class  Person
{       char  name[30];
        char address[50];
        int age;
        ---------
};
```

```
class  Student :  public  Person
{       char   collegeName[40];
        char   branch[20];
        -------------
};
```

The class Student inherits the properties of the class Person. The Person class is referred to as the base class, while the Student class is known as the derived class. In modern terminology, the base class is also called the parent class, and the derived class is also called the child class. All properties of the parent class are inherited by the child class. We can access the parent class's properties as local members in the child class, except for private properties. We know, private properties are accessed using public interface functions.

We will discuss inheritance concept in more detail in the next chapters.

---

static data member

If any variable is declared inside class scope then it reflects in every object we created, these are called instance or object variables, whereas static members create only once and shared to all objects.
That is, only one copy is created and shared to all objects of such class. Actually, its behavior is same as global variable in C. The main difference is, we declare them inside class scope.

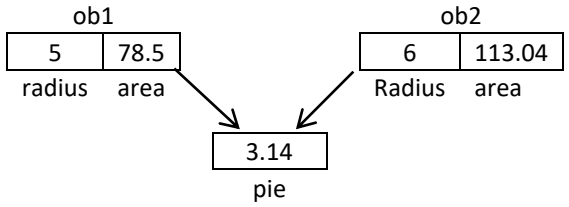
static members should be declared two times like given below program, first declaration should be inside of class and second declaration should be outside of class. The first declaration tells the variable is a class-member, and second declaration tells to the compiler to create space for static variable like global variable.

```
class Test
{       static  int  X;    // this is first time declaration, here space for 'X' will not be created(this is also called proto-type).
        ----------------   //  this declaration tells 'X' is a member of class
        ----------------
};
int Test::X=10;        // this is second time declaration, here space for X will be created with initial value 10.
```

As per C++ author version, these two declarations are said to be declaration & definition. Some people use these terms interchangeably, which creates confusion for others. If you are a beginner to programming, the definition and declaration of the term might be highly confusing. For avoiding confusion, modern programmers are using these two words as proto-type & declaration.

Let us see some practical examples, how static variables are shared among objects.

```
class  Circle
{       float radius, area;
        static  float  pie;    // only one copy of 'pie' creates for all objects of circle class (like global variable)
        --------------
        --------------
};
float  Circle::pie=3.14F;
int  main()
{      Circle  ob1(5) ,  ob2(6) ;
        --------------
        --------------
};
```



Different circles have different sizes of radius and area but all circles 'pie' value is same, so it is better to create only one copy and shared to all objects using static as shown above example.

---

more about static

Actually, static variables are used in two ways in the program, first style used to share them to all objects of the class, and second style is used as global variables or constant, however most of the time we use them in the first style only.

In case of first style use, we declare them in private section, so that, they can be accessed only in member functions of that class. As we have only one copy of static data, all member functions share same data.
In this way, all objects share same data of static through its member functions.

In case of second style use, we declare them in public section, so that, we can access them as global variable or global constant. Here the scope resolution operator(::) is required to access from outside of class. for example Math::pie (π), Color::green, Currency::rupees, etc.

Now you may get one doubt, what is the advantage of using static variable over global variable when a global variable concept is available in C++. Static variable gives more clarity than global variable and also avoids misusing. For example, the expression "Math::pie" tells that, we are accessing 'pie' value from Math class. So these are somewhat clarity with security.

Global and static variables behavior is same, that is, in compiler perspective there is no technical difference between them except place of declaration and accessing style.

When we come to accessing of static variable, if we are inside the class, that is, we are inside of member functions, the static variables can be accessed just as instance variables (scope operator is not required) but when we are outside of class, we have to access with class-name or object, but class-name is recommended. For example, Math::pie, Currency::rupees, etc.

Accessing static members with object is not all recommended, because other programmers may get confused whether it is static or non-static member. Let us see some examples one by one,

```
class Test
{       public:
        static  int  X;
        void show()
        {   printf(" %d  ",  X );  // inside member function, the static can be accessed just like instance variable.
        }
};
int Test::X=10;                 // here space creates for 'X' with initial value 10.

int   main()
{       printf(" output is %d " ,  Test::X );     // here class-name is required to access static from outside of class.
        Test  ob;
        printf(" output is %d " ,  ob.X );        // remember, this style of accessing with object is not recommended
        ob.show();      // output is: 10
}
```

Note: Remember, the expressions Test::X and ob.X access the same memory for X (since X is available as a single copy throughout the program). Therefore, all printf() statements in this program show the same output. Accessing a static variable through an object can indeed mislead programmers into thinking they are accessing an instance variable (non-static). So accessing with objet is not recommended.

Example: this example explains how static variable works as global.

```
class Test
{       public:
        static  int  X , Y;         //  this is proto-type of X,Y.
        void  increment()
        {       X++;    or   Test::X++;     // both are valid syntax, but "Test::X++"  is not recommended inside class
                Y++;    or   Test::Y++;
        }
};

int Test::X=10;     // this is declaration of static variable X, here space for X will be created with initial value is 10.

int Test::Y;          // this is declaration of static variable Y, here space for Y will be created with default  initial value is 0.

int  main()
{       printf("%d   %d",  Test::X ,  Test::Y );        //  10    0
        Test  ob;
        ob.increment();
        printf("%d   %d",  Test::X ,  Test::Y );        //  11    1
        Test::X++;  Test::Y++;
        printf("%d   %d",  ob.X ,  ob.Y );              //  12    2
        printf("%d   %d",  Test::X , Test::Y );         //  12    2
        printf("%d   %d",  X , Y );         //  error, undefined symbols X,Y. ( should be accessed with class-name or object)
}
```

---

static with constant

Example: static variables can be extended to constant variables, that is, we can make static members as constants too. Here only one copy will be created, and that too its value can't be changed. For example,

```
class Test
{       public:
        const static  int  X;     // static  with constant , value can't be changed
                static  int  Y;   //  only static, value can be changed
        ------------------------
        ------------------------
};
const  int  Test::X=10;    // we can't change this X value 10 in later part of the program
int  Test::Y=10;           // we can change this Y value in later part of the program
int main()
{   Test::X++;             // this is an error, because constant value can't be changed
    Test::Y++;             //  this is not an error, Y value can be changed
}
```

For example, the pie value of Circle is constant and does not need to change in the program, so it is better to declare it as static with the const type, as shown below

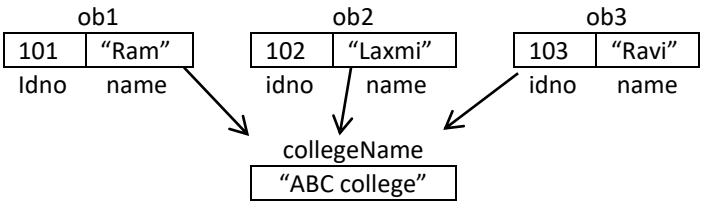
```
class  Circle
{       float radius, area;
        const  static  float  pie;   //  this is static as well as constant
        ----------------
        ----------------
};
const float  Circle::pie=3.14F;

int  main()
{       Circle::pie++;       // error, constant value cannot be changed
       ------------------
        ------------------
};
```

Example: the static variable must be declared two times inside & outside of class, otherwise an error

```
class Test
{       public:
        static int  X, Y;      //  this is proto-type declaration of X , Y
};
int   Test::X=100;             //  this second declaration is the actual declaration of variable.

int  main()
{       Test::X=200;
        Test::Y=300;           //  error, undefined symbol  Y ,  has not been declared outside of class
}
```

A practical example is a student class where the college name is the same for all students. In this case, it is better to declare the college-name as static to share it among all objects.

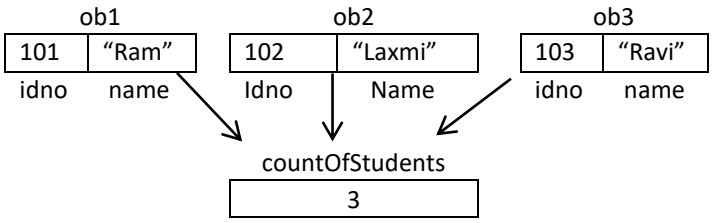
```
class Student
{       int  idno;
        string  name;
        static  char  *collegeName;
        ------------------------
        ------------------------
};
Student::char *p="ABC college";
```



Observe that, each student has its own idno, name, etc, but all students "collegeName" is same.
So it is better to declare "collegeName" as static for single copy.

Sometimes, we need to count the number of objects created for a class; for example, the number of clients connected to a server or the number of live viewers on a YouTube channel. Here, we use a variable like count with the static type to count the objects, as shown below.

```
class Student
{       int  idno;
        string  name;
        static  int   countOfStudents;

        Student()
        {       countOfStudents++;
        }
        -------------------
        -------------------
};
int  Student::countOfStudents=0;
```



Following examples explains how instance & static variables behaves in the program

```
class   Test
{       public:
        int  x;   static int  y;
        Test() { x=0; }
        void  add()
        {       x=x+10;  y=y+10;
        }
        void   show()
        {       printf("  %d  %d " , x  , y );
        }
};
int  Test::y=0;

int  main()
{       Test   ob1 , ob2, ob3 ;
        ob1.add();    ob2.add();   ob3.add();
        ob1.show();       //  10   30
        ob2.show();       //  10   30
        ob3.show();       //  10   30
```

Remember: In this program, 'x' has 3 copies in the form of ob1, ob2, and ob3 but 'y' has only one copy.

Initialization of static members in constructor

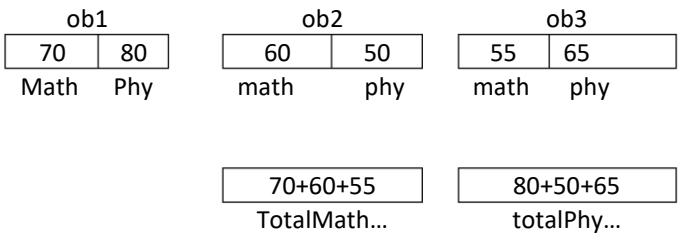
by mistake many people initialize the static members inside the constructor, it leads to improper result in the program. Let us see, following example.

```
class   Test
{       public:
        static int  x;
        Test() {  x=0;  }
        void    add()
        {   x=x+100;
        }
};
int  Test::x;

int   main()
{       Test   ob1;
        ob1.add();
        ob1.add();
        Test ob2;                   // here constructor will be called and makes the x to zero
        printf("%d  ",  ob1.x  );  //  output is : 0     ( not 200 )
}
```

we don't initialize the static member inside the constructor, otherwise re-initializes every time when new object is created. We expect the output as 200, but it shows 0

Complete the code of the following class yourself.

```
class  ExamTest
{   char  *name;
    int  math , physics;              //  stores each student marks of math and physics subject
    public:
    static  int  totalMathMarksOfAllStudents;     //  stores all students total marks of math subject
    static  int  totalPhysicsMarksOfAllStudents;  //  stores all students total marks of physics subject
    ------------

    ------------
};
```




```
int main()
{       ExamTest   ob1("Ram" , 70, 80 );
        ExamTest   ob2("Ravi" ,  60, 50 );
        ExamTest   ob3("Lamxi" , 55, 65 );
        ob1.show();
}
```

expected output will be: all students math total is : 165, and physics : 195

Hint: In the constructor add values to static members to get total values.

More about static data member

1) Static members also called ***class variables*** because they tied with class name instead of object.

2) The member functions of a class can be accessed static variables just like instance variables.

3) The non-member functions of class need to be accessed static with class-name by using scope resolution operator (for example Circle::pie). The member must be in public.

4) The non-member functions can also be accessed with object however accessing with class name is more recommend than object, so that, we don't get any confusion over accessing instance or static member.

5) Static data members create only once like global variable in C. These create before start of execution of main() function just like global variable. These variables behaviour is same as global variable in C. When many global variables exist then we get confusion which variable belongs to which part of the program, and a lot of misusing may happen in global variables (We may use one variable in place of another variable)

   Whereas static variables classified using class-name, therefore avoids confusion, avoids misusing and easy to remember, and use them. For example Math::pie, Colors::red, (instead of pie, red). finally, static makes more flexibility than global variables.

6) As we know, static should be declared two times in the program, this seems to be complex and confusion syntax. This is one of the main drawback in c++ syntax.

static member function

We know the class data reflects in the form of objects. To print or process such data, we call the member functions with the object, like ob.show(), ob.calc(), ob.scan(), etc. The function call ob.show() prints the data in 'ob'. So, without an object, the member function cannot print the data. Thus, we can say that member functions are tied to objects.

A static function, on the other hand, can only access other static members and not instance members. It is tied to the class name rather than an object. Static functions have independent behavior and can be called directly using the class name. If a function does not access instance variables, it is better to declare it as static.

```
class   Bank
{       int   balance;
        public:
        void showBalance()      // this function must be called with object, as 'balance' found in the objects
        {       printf(" balance is %d", balance );
        }
        static void showBankName()      // this fn is not accessing any data-member then static is the good choice
        {       printf(" SBI Bank, Vijayawada");
        }
};
int main()
{       Bank::showBalance();        // error, should be called with object
        Bank::showBankName();       // no error, this is the right way of calling static fn.
        Bank  ob;
        ob.showBalance();           // no error
        ob.showBankName();          // static function can also be called with object, but this is not recommended.
}
```

Note: static function can also be called with object, but this is not recommended. This calling makes confusion whether function is static or non-static function.

Accessing static and not-static

```
class   Sample
{   public:
    int  a , b ;
    static int c;
    void set1()
    {   a=10; b=20;     // no error
        c=30;           // no error
    }
    static void set2()
    {   a=10; b=20;     // error, static can access only other static
        c=30;           // no error
    }
};
```

```
int  Sample::c=0;
int main()
{   Sample ob;
    ob.a=100;
    ob.b=200;
    ob.c=300;
    Sample::a=10;       // error
    Sample::b=10;       // error
    Sample::c=10;       // no error
    Sample::set1();     // error
    Sample::set2();     // no error
    ob.set1();          // no error
    ob.set2();          // no error
}
```

Namespaces

Let there are two students with the same name in a classroom. To distinguish the two students, we need to call them by their names with surnames. A situation like this can occur in C++ as well. For example, if we write a function with name sqrt(), then it conflicts with the already existing library function sqrt() provided in math.h. In this case, the compiler cannot differentiate between the two function names. Hence, it will yield an error. To avoid this confusion, namespaces are used.

Namespace was introduced in second version of C++. They faced naming problem over **3rd** party libraries produced by different companies, conflicts happened over names of global variables, functions and classes. For solving this they had to develop namespace concept.

So namespace is a tool to resolve name conflicts in program libraries.
Namespace defines a named-block where we put variables, functions and classes.
These elements are accessed through scope-name with scope resolution operator(::)
Let us see an example

```
namespace  AA
{       int  x;
}
namespace  BB
{       int  x;
}
int  main()
{       AA::x=100;
        BB::x=200;
        printf(" %d  %d " ,  AA::x,  BB::x );    // 100   200
}
```

In this program, two global variables with the same name, 'x', will be created in different scopes.
The compiler allocates separate memory for these two variables as AA::x and BB::x. In this way, namespaces help resolve naming conflicts across multiple libraries.

Namespace can have functions, structures, and classes too. Let us see with functions

```
namespace   XX
{       void show()
        {       printf("hello from XX namespace" );
        }
}
namespace   YY
{       void show()
        {       printf("hello from YY namespace" );
        }
}
Int  main()
{       XX::show();     // op: hello from XX namespace
        YY::show();     // op: hello from YY namespace
}
```

Namespace can have at a time functions, classes and variables; Let us have one more example

```
#include<stdio.h>
namespace  ABC
{       int  K=200;
        void show()                     // it is global function like in C language
        {       printf("hello");
        }
        class Test                      // this is class Test
        {       public:
                static void show()
                {       printf("world");
                }
        };
}
int  main()
{       ABC::show();                    // hello
        ABC::Test::show();              // world
        printf("%d ", ABC::K );         // 200
}
```

This example explains with nested namespace

```
#include<stdio.h>
namespace XX
{       int a=100;
        static void show()
        {       printf("  hello  ");
        }
        class Test
        {       public:
                void show()
                {       printf("  world  ");
                }
        };
        namespace YY          // nested namespace
        {       float b=200;
        }
};
int main()
{       XX::Test  ob;
        printf(" output is %d " , XX::a );
        XX::show();
        ob.show();
        printf(" output is %.2f  " , XX::YY::b );
        return 0;
}
```

output is 100 hello world output is 200.00

‘using’ keyword in namespace

Note: if repeated accessing of namespace members are happened in the program, then accessing with scope resolution operator again and again becomes tedious, we can eliminate it by **‘using’** keyword. For example

```
#include<stdio.h>
namespace XX
{       int a=100;
        void show()
        {       printf("  hello  ");
        }
};
using  namespace  XX;                   // this eliminates the scope-resolution operator
int main()
{       printf(" a value is %d "  ,  a );       //  XX::  is not required before ‘a’
        show();                                 // this is same as XX::show();
        return 0;
}
```

---

syntax of namespace

```
namespace  namespace-name    // here “namespace” is a keyword
{       variables-list;
        functions-list;
        classes-list;
        nested-namespace-list;      // namespace can be nested
}
```

Points to note

We can have any number of namespaces in a program, and namespaces can also be nested.
Generally, namespaces are declared in the global scope; however, they can also be declared in a function's local scope, though only some compilers support this. Namespaces have many features, but they are not covered in this book.
The C++ compiler itself provides some libraries with namespace ‘std’ (also called the Standard Template Library). This namespace contained so many program elements. for example std::cin, std::cout, std::string, std::vector, std::list, std::array, std::queue, etc.
we will see more about this ‘std’ namespace in the next chapters.

C++ console i/o

For console i/o, we have used the printf() and scanf() functions in C. These can be used in C++ as well, but C++ people provided an alternative for a more type-less usage. They provided two predefined global objects called 'cout' and 'cin'. The cout object is used for printing values to the screen, and the cin object is used for reading values from the keyboard. Let's look at some examples before going deeper.

```
int  a=10;  float b=20;

cout<<a;                    //  this is equal to printf("%d", a );

cout<<b;                    //  this is equal to printf("%f", b );

cout<<"hello:"              //   printf("hello" );

cout<<"hello\nworld";       //   printf("hello\nworld");

cout<<"a value is "<<a;     //   printf("a value is %d", a );

cout<<a<<"  "<<b;           //   printf("%d  %f" ,  a , b );

cout<<" a is: "<<a<<" , b  is:"<<b;   // printf(" a  is: %d , b  is: %f " , a , b );

cin>>a;                     //   scanf( "%d", &a );

cin>>b;                     //   scanf( "%f", &b );

cin>>a>>b;                  //   scanf( "%d%f" , &a, &b );
```

Addition of two numbers

```
#include<iostream>
using namespace std;
int main()
{   int a,b,c;
    cout<<"enter two values:";
    cin>>a>>b;
    c=a+b;
    cout<<"output is :"<< c ;
}
```

ip: enter two values: 10 20 ↲
op: output is: 30

Scanning string and printing on the screen.

```
#include<iostream>
using namespace std;
int main()
{   char  a[100];
    cout<<"enter string::";
    cin>>a;
    cout<<" output is:"<<a;
}
```

ip: enter string: hello ↲
op: output is: hello

we can mix C & C++ i/o statements in a program

```
#include<iostream>
#include<stdio.h>
using namespace std;
 int main()
{   int a, b, total, avg;
    printf("enter two values: ");
    cin>>a>>b;
    total=a+b;
    avg=total/2;
    cout<<" total is:"<< total;
    printf(" average is %d", avg);
 }
```

ip: enter two values: 10 20 ↲
total is 30
average is 15

cin does not read multi word string.

```
#include<iostream>
using namespace std;
int main()
{   char  a[100];
    cout<<"enter string:";
    cin>>a;
    cout<<"output is:"<<a;
}
```

ip→ enter string: hello world ↲
op→ output is: hello (world did not scan)

to solve this problem, use: scanf(" %[^\n]", a);
or use getline() function (we will see later)

Let us come to more detailed.

- We have two API library classes called istream and ostream defined in the iostream.h file.
  These are stream classes used for input/output operations in C++.

- The 'istream' class has full of input functions for reading values from keyboard, while the 'ostream' class with full of output functions for printing values on the screen.

- Both classes consist of all overloaded functions for reading and writing all built-in types data, including int, float, char, char*, void*, etc.

- For each data type, there is a suitable overload function provided in these classes.

- The cin is an object of istream class, and the cout is an object of ostream class.
  These two objects defined globally in iostream.h file under the std namespace.

- We use stdio.h header file to use printf() and scanf()statements, similarly, iostream.h header file is used for cin and cout.

- In modern compilers the include statement should be in #include<iostream> (here .h not required)

- These are similar to scanf() and printf() functions in C, but the main difference is that cin and cout do not use formatted strings like %d, %f, %s, etc.

- As format strings are not required by cin and cout, these are particularly useful when working with template classes, where the types of values to be printed are unknown.

- We can use both c & c++ io statements at a time without any conflict. We can mix C & C++ io statements in a program.

- The cin object is linked with the **extractor** operator (>>), and the cout object is linked with the **inserter** operator (<<). The operator inserter(<<) inserts values into the output stream(monitor). That is, the inserted value reflects on the screen. The operator extractor(>>) extracts values from the input stream(keyboard) and assigns into given variables.

---

i/o manipulators in C++

setw() → this is same as C-Language format string “%5d”, %05d”, “%.2f” , etc.
sets width of the output value, it prints right-to-left on the screen (right justification),
if width > value then blank spaces are added before the value.

setfill() → it fills or prefixed specified character instead blanks.

endl → this is equal to next-line character(\n)

setprecision()→ used for floating point values. Prints specified fraction values.

fixed → show in normal style (fixed format)

scientific → shows in scientific notation (e power format)

oct → sets the base for the next output to octal (base 8).

hex → sets the base for the next output to hexadecimal (base 16).

The header file <iomanip> should be included to use these functions.

```
#include<iostream>
#include<iomanip>
using namespace std;
int main()
{   cout<<setw(5)<<123<<endl;
    cout<<setfill('*')<<setw(7)<<123<<endl;
    cout<<setfill('A')<<setw(7)<<123<<endl;
    cout<<setfill('@')<<setw(7)<<123<<endl;
    cout<<hex<<10<<11<<12<<endl;    // hexa decimal

    cout<<fixed<<setprecision(2)<<123.4556f<<endl;
    cout<<scientific<<setprecision(2)<<123.455f<<endl;
}
```

Output

```
  1 2 3
* * * * * 1 2 3
A A A A A 1 2 3
@ @ @ @ @ 1 2 3
A B C

1 2 3 . 4 5
1 . 2 3 e + 0 0 2
```

Some additional manipulators are

left: Aligns the next output to the left within the field width.

right: Aligns the next output to the right within the field width.

internal: Aligns the sign of the next output to the left within the field width, with any padding placed right.

dec: Sets the base for the next output to decimal (base 10).

oct: Sets the base for the next output to octal (base 8).

hex: Sets the base for the next output to hexadecimal (base 16).

showpos: Displays the sign of positive numbers in the next output.

noshowpos: Hides the sign of positive numbers in the next output.

showpoint: Forces the decimal point to be displayed for floating-point numbers, even if there are no decimal places.

noshowpoint: Hides the decimal point for floating-point numbers if there are no decimal places.

fixed: Sets the output format to fixed-point notation.

scientific: Sets the output format to scientific notation.

hexfloat: Sets the output format for floating-point numbers to hexadecimal.

defaultfloat: Resets the output format for floating-point numbers to the default format.

flush: Flushes the output buffer without inserting a newline character.

Operator overloading

Let us come to a simple addition on class objects, previously, we have added two objects like ob1.add(ob2) and compared like ob1.compare(ob2). We have written and called the member functions in that way.

Generally, we add two integers like 10+20. In the same style, if we add two class objects using the '+' operator, such as ob1+ob2, it would be a great choice for the programmer to handle class objects as easily as int and float. The operator overloading concept allows us to handle class objects in this way.

Let us take two types of addition: 10+20 and 10.45+59.34; the first one is integer addition, and the second one is float addition. In both cases, the same '+' operator is used. Even though both values are added using the same '+' operator, the compiler generates different machine code for different data types, because the machine code for integer addition is not the same as for float addition. In this way allowing same '+' operator for different types of data is said to be operator overloading, and this '+' operator is overloaded at the compiler level to support all built-in types of data. Of course, we have been using the '+' operator without this knowledge from the first day. The compiler manufacturers overloaded all operators on all built-in type data. This is also called functional abstraction at the compiler level.

In C++, this concept is expanded to allow programmers to create custom type additions on objects. Not only can we perform addition, but we can also do all arithmetic, relational, and logical operations. Previously, we have added two objects like ob1.add(ob2) and compared like ob1.compare(ob2). Now, using the operator overloading concept, we can add as ob1+ob2, and we can compare as ob1==ob2, etc. This is 100% similar to operations on integer like 10+20 and 10==20. In this way, using the operator overloading concept, we can handle our class objects just like any int and float types. ***The main objective of operator overloading is to create our own class-based user-defined data types that can be used similar to built-in types.*** This comes under polymorphism concept, where one interface can take many forms.

Operator overloading is a good choice for **frequently used objects**, such as Time, Date, DateTime objects. However, for normal use objects, the call expression ob1.add(ob2) is sufficient instead of ob1+ob2. For frequently used objects, like those in API class libraries, operator overloading would be a better choice.

The term 'operator overloading' is a short name for operator function overloading. This is very similar to the concept of function overloading. **Actually, the operator overloading is nothing but a function with a special syntax.** While writing the function body, we use an operator symbol in place of function name. Here, the operator symbol acts as the function name. Syntax for operator-function is,

```
return-type   operator <symbol> ( parameters list )   // here symbol can be any + , - , * , < , > , etc
{       ---------
        ---------
        return value;
}
```

```
eg:     Time  operator + ( Time ob2)
        {       ---------
                ---------
                return value;
        }
```

here **"operator+"** acts as a function-name

You might be wondering how to call this function. Calling this function is the same as using mathematical expressions like ob1 + ob2, ob1 - ob2, ob1 * ob2, ob1 < ob2, ob1++, etc.

The compiler converts the expression ob1+ob2 into a function call statement as ob1.operator+(ob2). Here the word 'operator' is a keyword added by the compiler as a function-name with the symbol '+'. Thus the "operator+" works as a function-name in the program.

The instruction ob1+ob2 converts into function call as: ob1.operator+(ob2); // this is like ob1.add(ob2)
The instruction ob1+100 converts into function call as: ob1.operator+(100); // this is like ob1.add(100)

Let us have one example, adding two time objects using symbol '+' as

```
int main()
{       Time ob1(4, 5, 20), ob2(3, 15, 50), ob3;
        ob3=ob1+ob2;   → ob3 = ob1.operator+(ob2); , remember this is a function-call statement
        ob3.show();
}
class Time
{       ------
        Time operator+ ( Time ob2 )      // this is a function-body. Observe that function name is 'operator+'
        {   Time t;
            ---------
            return t;
        }
        ----------
};
```

Addition of two players score in a game using "operator+" function.

```
int main()
{       Player ob1(10) , ob2(20), ob3;      // ob1, ob2 holds two player's score. Initially taken as 10, 20
        ob3=ob1+ob2;   → ob3=ob1.operator+(ob2) , remember this is a function-call
        ob3.show();
}
class  Player
{       private: int score;
        public:   Player() { score=0; }
        Player( int s ) { score = s; }
        Player operator+(Player ob2)
        {   Player t;
            t.score=score+ob2.score;
            return t;
        }
        void show()
        {   printf("score is %d", score);
        }
};
```

```
ob3 = ob1 + ob2;
        ↓
ob3 = ob1.operator+(ob2);          // function call
        ↓
Player operator + (Player ob2)     // function-body
{       Player t;
        t.score = score + ob2.score;
        return t;
}
```

Let us see how following expressions are converts into function-call statements.

ob1 + ob2 → ob1 . operator + (ob2)
ob1 + 100 → ob1 . operator + (100)
100 + ob1 → operator + (100 , ob1)
ob1 * 100 → ob1 . operator * (100)
ob1 / 100 → ob1 . operator / (100)
ob1 < 100 → ob1 . operator < (100)
100 < ob1 → operator < (100 , ob1)
++ob → ob.operator++();

Finding addition of two times which is employee worked in 2 shifts, and also adding seconds to time.

ob3=ob1+ob2; → this instruction adds two time objects ob1 & ob2 into ob3
ob3=ob1+7200; → this instruction adds ob1 & 7200 seconds into ob3

```
#include<stdio.h>
class  Time
{       int  h,m,s;
        public:
        Time()  { h=m=s=0; }
        Time(int h, int m, int s)  { this->h=h;    this->m=m;  this->s=s; }

        Time  operator+( Time  ob2)
        {       Time t;  int x;
                x= (h+ob2.h)*3600 + (m+ob2.m)*60 + (s+ob2.s);
                t.h = x/3600;
                t.m = (x%3600)/60;
                t.s = (x%3600)%60;
                return t;
        }

        Time  operator+( int x )
        {       Time t;
                x=x+h*3600 + m*60 + s;
                t.h = x/3600;
                t.m = (x%3600)/60;
                t.s = (x%3600)%60;
                return t;
        }

        void  show()
        {       printf("\n output is %d : %d : %d", h,m,s);
        }
};

int  main()
{       Time   ob1(5,20,30) , ob2(3,40,50) , ob3;
        ob3=ob1+ob2;     // ob3=ob1.operator+(ob2);
        ob3.show();
        ob3=ob1+7200;    // ob3=ob1.operator+(7200);   adding 7200 is seconds
        ob3.show();
}
```

More about function return-value

Before returning a value to the calling function, it is stored in a temporary variable and then returned.
let us see following example

```
int add(int a,int b)
{
        return a+b;
}
```

```
int  main()
{       int k;
        k=add(10,20);
        cout<<k;
}
```

Here before returning a+b value to main() function, the compiler stores the a+b value into a temporary variable and then returned. Actually, this temporary variable is a nameless variable created & destroyed by the compiler. For a while, let us say its name is 'temp'. The data type of 'temp' is depends upon a+b result type. The above function-body can be imagined with 'temp' as:

```
int  add( int a, int b )
{       int temp;
        temp=a+b;
        return temp;
}
```

In this way compiler creates and destroys temporary space when the function has return-value.

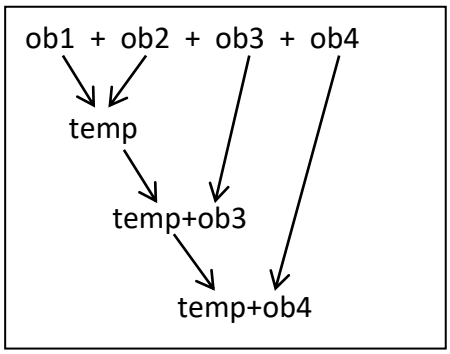
Adding multiple objects

sometimes we may need to add more than one object at a time like ob1+ob2+ob3+ob4. Then the compiler converts this statement into function calls as

```
total=ob1+ob2+ob2+ob4;
```

↓

```
temp=ob1+ob2;
temp=temp+ob3;
temp=temp+ob4;
total=temp;
```



Here 'temp' variable is nothing but a variable that holds the return value of the function, like in the example said above. The 'temp' is used to hold the return value and also to add with the next object

Let us see following example on this concept

In a school there are boys and girls, now take 3 schools data and print total count of boys & girls separately.

```
#include<stdio.h>
class  School
{       private:  int boys , girls;
        public:
        School()
        {   boys=girls=0;
        }
        School(int boys , int girls)
        {   this->boys=boys;
            this->girls=girls;
        }
```

```
    School operator+(School ob2)
    {       School t;
            t.boys=boys+ob2.boys;
            t.girls=girls+ob2.girls;
            return t;
    }
    void show()
    {       printf("boys count = %d", boys);
            printf("\ngirls count = %d", girls);
    }
};
int  main()
{       School ob1(10,20) , ob2(30,40), ob3(50,60);
        School  ob4;
        ob4 = ob1 + ob2  + ob3;
        ob4.show();
}
```

temp=ob1+ob2; → temp=ob1.operator+(ob2);
temp=temp+ob3;→ temp=temp.operator+(ob3);
total=temp;

BADMAS rules in operators.

Compiler follows BADMAS rules as in mathematics. For example, ob1+ob2*ob3; here first multiplication operator is executed before addition. Similarly parenthesis () also works on objects to change the priority.

```
#include<stdio.h>
class  Player
{       int  score;
        public:
        Player() { score=0; }
        Player( int s ) {  score=s; }

        Player  operator + (Player ob2)
        {       Player t;
                t.score = score + ob2.score;
                return t;
        }
        Player operator * ( Player ob2)
        {       Player t;
                t.score=score*ob2.score;
                return t;
        }
        void  show()
        {       printf("\n score is %d ", score);
        }
};
int  main()
{       Player ob1(2), ob2(3), ob3(4), total;

        total=ob1+ob2*ob3;
        total.show();   → score is 14

        total=ob1*ob2+ob3;
        total.show();   → score is 10

        total=(ob1+ob2)*ob3;
        total.show();   → score is 20
}
```

Comparison of two dates using equal operator(==)

```
int  main()
{       Date  ob1(10,4,2010),  ob2(12,4,2022);
        if( ob1==ob2)                    //  if( ob1.operator==(ob2) )
                cout<<"equal";
        else
                cout<<"not equal";
}

class  Date
{               int  d,m,y;
        public:
                Date( int d, int m, int y)
                {       this->d=d;   this->m=m;  this->y=y;
                }
                bool  operator==( Date  ob )
                {       if( d==ob.d  && m==ob.m && y==ob.y)
                                return true;
                        else   return false;
                }
};
```

Adding two complex numbers using operator overloading (+).

```
class  Complex
{           int  real, img;
        public:
            Complex ( int real , int  img )
            {       this->real=real;   this->img=img;
            }
            Complex   operator + ( Complex  ob )
            {       Complex  t;
                    t.real=real+ob.real;
                    t.img=img+ob.img;
                    return t;
            }
            void show()
            {       cout<<"output is: "<< real << " + " << img<<"i"<<endl;
                    // printf("output is:  %d + %d i ",  real, img);
            }
};

int  main()
{           Complex   ob1(10,20),  ob2(5,7),  ob3;
            ob3=ob1+ob2;
            ob3.show();
}
```

Complete the following Date class, which handles operations such as adding days, subtracting days, comparing days, and printing. The main() function is provided along with sample data to test the program.

```
class Date
{       private:  int  d, m, y;
         public:  Date() {…}
                  Date ( int  d, int  m, int  y) { … }
                  Date  operator+( int n) { … }          // increment date by n-days
                  Date  operator-( int n) { … }          // decrement date by n-days
                  bool  operator==( Date) { … }          // check two dates are equal or not
                  bool  operator<( int days) { … }       // compare whether date1<date2
                  bool  operator>( int days) { … }       // compare whether date1>date2
                  bool  operator ! () {…}                // checks whether date  is valid or  not
                  void   show() { … }
};
int  main()
{       Date ob1(10,5,2020),  ob2(12,4,2020), ob3;
        ob3=ob1+365;
        ob3.show();
        ob3=ob1-365;
        ob3.show();

        if( ob1<ob2)  cout<<" Date1 < Date2 ";
        else  cout<<" Date1 < Date2 ";

        if( ob1==ob2)  cout<<" two dates are equal ";
        else  cout<<" two dates are not equal ";

        if( !ob ) cout<<" in valid date ";
        else cout<<" valid date ";
}
```

Complete the following Time class, which handles operations between two time objects such as addition, subtraction, comparison, and printing. The main() fn is provided along with sample data to test the program.

```
class Time
{       private:  int  h, m, s;
        public:   Time() {… }
        Time( int  d, int  m, int  y) { … }
        Time  operator+( int n) { … }          // increment Time by n seconds
        Time  operator-( int n) { … }          // decrement Time by n seconds
        bool  operator==(Time) { … }           // check two Times are equal or not
        bool  operator<( Date ob2) { … }       // compare whether ob1<ob2
        bool  operator>( Date ob2 ) { … }      // compare whether ob1<=ob2
        void   show() { … }
};

int  main()
{       Time  ob1(4,5,40), ob2(2,4,50), ob3;
        ob3=ob1+7200;   ob3.show();
        ob3=ob1-7200;   ob3.show();
        if( ob1<ob2)  cout<<"time1<time2";
        else if( ob1>ob2)  cout<< "time1>time2";
        else if( ob1==ob2)  cout<<"two times are equal";
        else  cout<<"two times are not equal";
}
```

Overloading ++ and – – operators

We can overload the ++ and -- operators on our class objects just like other operators.
For example: ++ob and ob++. The compiler converts these expressions as shown below:

++ob → ob.operator++()
ob++ → ob.operator++(0) // here, zero is a dummy-value used to differentiate pre & post increments.

The pre-increment instruction ++ob is converted into function call as ob.operator++(). Observe that the function name here is 'operator++'. Inside this function body, we can write any code just like in a normal function.
Whereas the post-increment instruction ob++ is converted into a function call as ob.operator++(0). Here, the argument zero is added by the compiler as a dummy argument to differentiate between pre-increment and post-increment.

Let us consider a Player class where we increment the player's score by 1. This example demonstrates both pre-increment and post-increment.

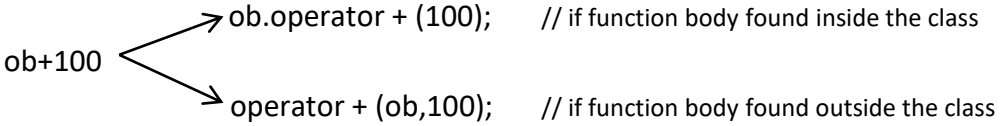
```
class  Player
{          int score;
       public:
          Player() { score=0; }
          Player( int s) { score=s; }

          void   operator++()                  // this function calls for pre-increment
          {      score=score+1; or  score++;
          }
          void   operator++( int dummy )   // this function calls for post-increment. Here dummy value does nothing.
          {      score=score+1;   or score++;
          }

          void show()
          {      cout<<"player score is : "<< score;
          }
};

int   main()
{      Player ob(10);
       ++ob;     //  ob.operator++();
       ob.show();
       ob++;     //  ob.operator++(0);
       ob.show();
}
```

The function operator++() increments the player's score by 1 point. This function currently returns nothing. However, sometimes this function needs to return the incremented object so that it can be assigned to another object. For example: ob2=++ob1; here, the incremented value of ob1 is assigned to ob2. In this case, the function should be written as

```
Player   operator++( int dummy )
{      score=score+1;
       return *this;    // here 'this' pointer is pointing  to object 'ob', so the expression "*this" returns the copy of ob.
}
```

Overloading operator inside or outside of class

we can write the operator overload function either inside or outside of class. Compiler gives the programmer the freedom to choose either style. Based on where the function body is written in the program, compiler converts the function call statement accordingly, as shown in the styles below.



The compiler converts based on the function body found in the program, whether it is a member or non-member function. If the function body is written inside the class body, it converts using the first style. If the function body is written outside the class body, it converts using the second style. It is fairly depends on you. Actually, the first style of writing functions as member functions inside the class makes the code more flexible and simpler(compiler always looks for this style only) . The second style is used when left hand side operand is not class object like 100+ob, we will see this in next topic

Operator function inside the class body

ob + 100

↓

ob.operator + (100)

↓

```
class  Player
{   public:   int score;
              Player() { score=0; }
              Player operator+(int n)     // class inside
              {       Player t;
                      t.score = score+n;
                      return t;
              }
              void show()
              {   cout<<score;
              }
   };

   int main()
   {       Player  ob;
           ob=ob+100;    // ob=ob.operator+(100);
           ob.show();
   }
```

Operator function outside the class body

ob + 100

↓

operator + (ob , 100)

↓

```
class  Player
{   public:   int score;
              Player()  { score=0; }
              void show()
              {   cout<<score;
              }
};
Player operator + (Player ob , int n)  // class outside
{       Player t;
        t.score = ob.score+n;
        return t;
}
int main()
{       Player  ob;
        ob=ob+100;   // ob= operator+(ob,100);
        ob.show();
}
```

In the above program, the data member 'score' is declared as public to make the code easier to understand. Generally, we put data members in private section for security. However, if we do that, a non-member function defined in the second style cannot access private data. To solve this problem, we declare it as a friend function (revise the concept of friend functions). The following is an updated version with a friend function.

```
class  Player
{     private:
            int score;
       public:
            Player() { score=0; }
            void show()
            {     cout<<score;
            }
            friend Player operator + ( Player, int );   // now this function is friend to class, so it can access private data.
};

Player operator + (Player ob , int n)
{      Player t;
       t.score = score + n;
       return t;
}

int  main()
{      Player  ob;
       ob=ob+100;    // ob=operator+(ob,100);
       b.show();
}
```

Overloading when the left-hand side operand is not object.

ob+10 → here the left hand side operand 'ob' is class-object. So compiler takes as: ob.operator + (10)
10+ob → here the left hand side operand '10' is not class object. So compiler takes as: operand+(10,ob)
Generally, the member functions of a class are called with object like ob.show(); whereas the non-member functions are called like show(); Here **10+ob→operator+(10,ob)** is a normal function call. So the function body should be written outside class as a non-member-function. Let us take 'Player' class objet

ob + 10 ↓ ob.operator + (10); ↓ Player operator + (int n) { ------ ------ }	10 + ob; ↓ operator + (10 , ob); ↓ Player operator + (int n , Player ob) { ------- ------- }

Following example explains operator+() function with & without left-hand-side operand by class-object. Example on Player class objects, where adding score to object like ob+5 and 5+ob.

```
int main()
{       Player ob1(10) , ob2 ;
        ob2=ob1+5;          // ob2 = ob1.operator+(5);
        ob2.show();

        ob2=5+ob1;          // ob2 = operator+(5, ob1);
        ob2.show();
}
class  Player
{       public: int score;
        Player() { score=0; }
        Player( int score ) { this->score=score; }
        Player operator+( int s )
        {       Player temp;
                temp.score = score + s;
                return temp;
        }
        void show()
        {       cout<<"\n score is :"<<score;
        }
};
Player operator+( int s , Player ob )
{       Player temp;
        temp.score = ob.score + s;
        return temp;
}
```

As per oop concept, everything which belongs to class object should be written inside class such data-members and member-functions, but C++ compiler manufacturers broke this rule and asked us to write this function outside class. C++ compiler itself did not follow oop rules in some cases such as static-data-members, C global functions, namespaces, etc. Thus C++ is not pure object oriented language. However, many people in the world like C++ as it introduced many new concepts in oop programming.

In the above program, the data member 'score' is kept in the public section to allow access from outside the class in a non-member function. However, doing so is not considered good programming practice. It is better to place it in the private section and access it through a friend function. (See friend function concept)

Let us see modified code above class.

```
class   Player
{       private:  int  score;
         public:  -------
                  -------
        friend  Player  operator( int n ,  Player );
};

Player  operator+( int s , Player ob )          // making this function as friend to above class.
{       Player  temp;
        temp.score = ob.score + s;
        return temp;
}
```

This is one more example explains “operator+()” function with & without left-hand-side operand by class-object. Explained with Time class, here 7200 seconds(2hours) are adding to the given Time object.

```
int  main()
{       Time  ob1(2, 10, 20) ,  ob2;

        ob2 = ob1 + 7200;       //  ob2=ob1.operator + (7200 );
        ob2.show();

        ob2 = 7200 + ob1;       //  ob2=operator + ( 7200, ob1 );
        ob2.show();
}
class  Time
{       private:   int h, m, s;
        public:   Time() { }
                  Time(int h, int m, int s)
                  {       this->h=h; this->m=m; this->y=y;
                  }
                  Time operator+( int n )
                  {       Time  t;
                          n=n+h*3600 + m*60 + s;
                          t.h=n/3600;
                          t.m=(n%3600)/60;
                          t.s=(n%3600)%60;
                          return t;
                  }
                  void  show()
                  {       printf("\ntime is: %d-%d-%d" , h, m, s );
                  }
                  friend Time operator+(int n, Time ob);
};
Time operator + (int n, Time ob)
{
        return ob+n;                    // this instruction again calls the member function for ob+n
}
```

Overloading minus sign(-) operator

For example, the expression **ob2 = -ob1** is convert into function call statement as **ob2=ob1.operator-()**
Here we need to overload the minus operator as shown example below .

```
#include<iostream>
class Test
{       int a;
        public:
        Test() {  a=123; }
        Test operator-()
        {       Test ob;
                ob.a=-a;
                return ob;
        }
        void show()
        {       printf("%d ", a);       //  output is: -123
        }
    };
    int main()
    {       Test ob;
            ob=-ob;          //  -ob  →  ob.operator-()
            ob.show();
    }
```

==========

Overloading type-cast operator

From above knowledge, the compiler takes following expression as

```
Time ob;   int k;
ob+10  → ob.operator+(10)       //  observe,  left hand side operand is class object.
10+ob  → operator+(10,ob)       //  observe,  left hand side operand is not class object
ob=10 → ob.operator=(10);       //  left hand side is class object
 k=ob  → operator=(k , ob);     //  left hand side is not class object.
```

In the last expression, “k = ob”, we expect it to be an assignment operator overloading (=) and interpret it as a function call statement like ‘operator=(k,ob)’. Particularly for this type of expression, the compiler does not take as we expected. In this special case, the compiler treats it as a **type conversion** operator, which requires us to overload it as shown below

k = ob → we understand it as k = (int)ob;

therefore, the expression (int)ob → converts into a function call as → ob.operator int()

```
ob.operator  int()
    ↓
operator  int()     // this function does not have return-type, here ‘int’  itself works as return-type.
{   ----
    ----
}
```

let us see how following type-cast expressions are converted into a function-call statements as

(int) ob → ob.operator int()

(float) ob → ob.operator float()

(char*) ob → ob.operator char*()

Following class 'Test' gives an idea how to return an integer value from an object using type-cast operator.

```
class Test
{   private: int a;
    public: Test() { a=100; }
            operator int()      // does not have return-type. Here 'int' itself works as return-type.
            {   return a;
            }
};
int   main()
{       Test ob;  int k;
        k = ob;           //  k = ob.operator int();
        cout<<"\noutput is:" << k ;
        return 0;
}
```

The expression k = ob; in the main() function can also be written as **k = (int)ob;**

but the **k=ob** is simple and enough, because, compiler automatically translates this into **k=(int)ob.**

Based on the function-body of operator-cast, the compiler automatically translates in this way.

Following class 'Test' gives an idea how to return a string value from an object using type-cast operator.

```
class Test
{       private:
            char *a;
            Test()  { a="hello";  }
            operator  char*()     // this function does not have return-type.
            {   return a;         // Remember, return type of this function is same as "char*"
            }
};
int  main()
{       Test ob;    char *p;
        p=ob;           // ob.operator  char*();
        cout<< "output is :" << p ;
        return 0;
}
```

Complete the following code of Date class. Here main() function body provided with some function calls, based on this calls, implement all functions as needed in the program.

```
class   Date
{   -------
    -------
};
int  main()
{   Date  ob;   int  n = 12032024;      // 12-03-2024
    ob = n;
    ob.show();       // Date is: 12-03-2024
    int  n=ob;
    cout<<"\n date in integer format is:"<<n;     // date in integer format is: 12032024
}
```

Complete the following code of Time class. Here main() function body provided with variety of function calls, based on this calls, implement all functions as needed in the class.

```
int  main()
{       Time  ob1(2, 10,20), ob2, ob3;
        ob1  = 7290;         //  ob1 = 2hr, 1min, 30sec
        ob2 = 9870;          //  ob2 = 3hr, 1min, 10sec;
        ob3 = ob1 + ob2;
        ob3.show();
        ob3 = 3600+ob1;
        ob3.show();
        int a = ob1;      //  convert time into seconds and return it.
        cout<<" time in second is:"<<a;    //  7290
}

    class  Time
    {       private: int  h, m, s;
             public:  --------
                      --------
    };
```

Complete the following class 'String' using static & dynamic arrays. Here main() function provided with variety of string function calls, based on this code, implement all string functions as needed.

```
int  main()
{       String   ob1("hello"),  ob2("world"),  ob3;
        ob3 = ob1 + ob2;
        ob3.show();

        ob3 = ob1 + " Hi";
        ob3.show();

        ob3 = "hello" + ob1;
        ob3.show();

        if( ob1 == "hello" ) cout<<"equal";
        else cout<<"not equal";

        ob1 = "AAA";
        ob2 = "BBB";
        ob3 = ob1 + ob2;
        ob3.show();
        char *p;
        p = ob1;
        cout<<"output is :"<<p;         // AAA
}
```

```
class  String                          class  String
{       private: char  a[100];         {       private:  char  *a;
        ------                                 ------
        ------                                 ------
};                                     };
this is using static-arrays            this is using dynamic-arrays
```

Overloading () and [] operators.

Even we can overload parenthesis operators in C++, used especially while handling array-list type values.
For example, the expression "ob(0)" converts into a function call statement as "ob.operator() (0)".
Let us see some expressions, how they are converted into function call statement

ob(2) → ob.operator() (2); // here name of function is "operator()"
ob(k) → ob.operator() (k);
ob() → ob.operator() ( );

For easy understanding, let us take our well known example "Date class"
for example ob(1) returns day in a date, ob(2) returns month in a date, etc.

```
Date  ob(10, 5, 2024);
int day, month, year, k;
day=ob(1);              // this function returns day in a date.
month=ob(2);            // this function returns month in a date.
year=ob(3);             // this function returns year in a date.
k=ob();                 // this function returns total date in integer format  12-4-2024 → 12042024

#include<iostream>
#include<string.h>
#include<stdio.h>
using namespace std;
class Date
{       private:  int d,m,y;
        public:

        Date( int d, int m, int y )
        {       this->d=d;  this->m=m; this->y=y;
        }

        int  operator() ( int k=0 )          // zero is the default argument
        {       if(k==1) return d;
                else  if(k==2) return m;
                else  if(k==3) return y;
                else  return d*1000000+m*10000+y;
        }
};

int main()
{       Date ob(12, 5, 2024);
        cout<<"\n day in a date is:"<<ob(1);
        cout<<"\n month in a date is:"<<ob(2);
        cout<<"\n year in a date is:"<<ob(3);
        cout<<"\n date in a number is:"<<ob(0);
        cout<<"\n date in a number is:"<<ob();          // default argument  is  zero
        return 0;
}
```

Following program explains Array-List class, it assigns values to object and retrieves the values based on index. Here two operators are over-loaded assignment operator(=) and parenthesis operator ().

```
#include<iostream>
using namespace std;
class  ArrayList
{       private :
                int  *a,  count,  size;
         public :
                ArrayList ( int  size )
                {       this->size=size;
                        count=0;
                        a=new int[size];
                }
                int operator() ( int  i )
                {       if(i>=count)
                        {       cout<<"\n invalid index";
                                return -1;
                        }
                        return a[i];
                }
                void show()
                {       cout<<"\n output is:";
                        for(int i=0; i<count; i++)
                            cout<<a[i]<<" ";
                }
                void operator=( int v )
                {       if( count >= size )
                        {       cout<<"\n array full , can't be inserted";
                                return;
                        }
                        a[count++]=v;
                }
};
int   main()
{       ArrayList ob(10);
        ob=10;              // ob.operator=(10);  adds at end of list
        ob=20;
        ob=30;
        ob.show();
        int  k=ob(1);       //  k=ob.operator() (1)
        cout<<"\nvalue at a[1] is:"<<k;
}
```

Overloading assignment operator(=) and copy-constructor

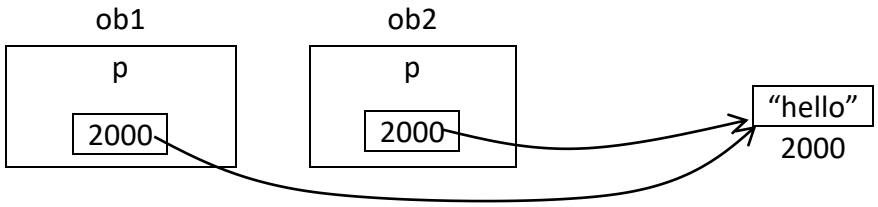
Let us know need of overloading assignment operator

For example, the addition of two class objects like ob1+ob2 cannot be understood by the compiler, because complier does not know how to add two object's data. If two objects are 'time-class' type, they **cannot be** added just as two integers. So here we need to write an overloaded function to add two objects. This is what we learned previously.

But when we come to assignment of one object's data to another object like ob2=ob1, the compiler copies bit by bit of ob1 data to ob2. So here overloading assignment operator is not required. However, if object contained a pointer and it is pointer to dynamic memory or outside object memory then it creates a problem. Here after assignment, the both pointers in objects may points to same memory. For example,

```
String  ob1("hello")  , ob2 ;
ob2 = ob1;
```

after this assignment, both pointers in ob1 & ob2 are pointing to same memory like below picture



If the string 'hello' is changed through 'ob1', it also affects 'ob2' because, after the assignment ob2=ob1, both ob1.p and ob2.p pointers are pointing to the same memory (same string).

From the compiler's perspective, when ob2 = ob1 is executed, the compiler copies only member to member within the objects but not the outside object's memory (the compiler does not care about outside pointer pointing memory). Therefore, it only copies the address 2000 of ob1.p to ob2.p. As a result, both pointers point to the same memory (the same string, "hello"). If the string is changed through ob1.p, it also affects ob2.p. The following example explains how one object affects the other when pointing to the same memory.

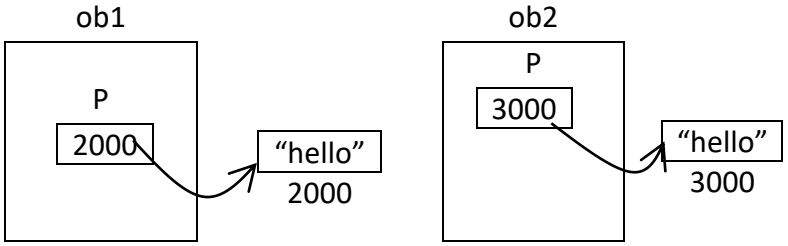
```
#include<stdio.h>
#include<string.h>
class String
{       private:  char  *p;
        public:   String()
                  {       p=NULL;
                  }
                  String( char *p)
                  {       p=new  char[100] ;
                          strcpy( this->p , p );
                  }
                  void change( char *p )
                  {       strcpy( this->p , p );
                  }
                  void show()
                  {       printf("\n output is %s" , p );
                  }
};
```

```
int  main()
{       String  ob1("hello") , ob2;
        ob2=ob1;                        // address is copied to ob2,  not  'hello'
        ob1.change("world");        // if ob1 is changed then it also affects to ob2.
        ob1.show();     → world
        ob2.show();     → we expect "hello"  but  shows  "world" → this is wrong output
}
```

Like above said, when ob2=ob1 is executed then compiler copies ob1.p to ob2.p (but not string "hello") Therefore, the function call ob1.change("world"), changes the string "hello" to "world", affecting both ob1 & ob2. The solution is to overload the assignment operator(=) by creating a separate memory for ob2.
The function is as follows

```
        ob2=ob1;
          ↓
    ob2.operator=(ob1);
          ↓
void operator=( String  ob1 )
{       if(p==NULL)
            p=new char[100];     // here creating separate memory for ob2.
        strcpy( p, ob1.p );
}
```

Add this function to the class so that the compiler calls it when an assignment occurs between objects.
Here this function creates separate memory for ob2 and copies the string, as shown below.



However, this function does not work for multiple assignments, for example, ob3=ob2=ob1; because the function simply assigns ob1 to ob2 and returning nothing, here the return type is void. So the above function should be re-written as

```
String  operator=( String  ob1)
{       if(p==NULL)
                p=new char[100];
        strcpy( p , ob1.p );
        return ob1;         // return of this ob1 is assigned to ob3
}
```

Copy constructor

```
String  ob1("hello");
String  ob2(ob1);       // this instruction can also be written as: "String ob2=ob1";
```

String ob1("hello"); → As we know, compiler cannot initialize "hello" to ob1. It does not know how to put "hello" to ob1. We need to write a constructor to copy "hello" to ob1. This is what we did previously.

String ob2(ob1) → Here, the compiler automatically initializes (copies bit by bit) from ob1 to ob2. So we don't need to write any constructor for this. However, if object contains a pointer and that is pointing to external memory (other than object itself), the same problem as mentioned earlier will arise.(like ob2=ob1) To solve this problem, we need to write our own version of the constructor to copy one object's data to another. This is often called the copy constructor. (A copy constructor means: it copies/initializes one object with another object).
Many people get confused about the assignment and initialization of objects. Let us see the code

1)
```
String ob1("hello"),  ob2;
ob2=ob1;                    // this is an assignment, assignment operator function will be called
```

2)
```
String  ob1("hello");
String  ob2(ob1);           // this is an initialization, not assignment, constructor will be called.
```

The instruction **"String ob2(ob1)"** can also be written as **"String ob2=ob1";**

String ob2(ob1) or String ob2=ob1 → this instruction may seem like assigning ob1 to ob2, and many people misunderstand it in this way. Actually, this is initializing ob2 with ob1. The copy constructor will be called.

String ob2(ob1) → this statement can be consider as two instructions **"String ob2"** and **"ob2=ob1";**
The first one **"String ob2"** is said to be declaration of object, here memory space creates for ob2.
The second one **"ob2=ob1"** initializes ob2 with ob1. Remember, assigning values at the time of declaration of variables is said to be initialization and not assignment.

Following program explains without overloading copy-constructor, and see how ob2 disturbs the ob1.

```
#include<stdio.h>
class String
{           char  *p;
        public:
            String() { }
            String ( char *p )
            {       p=new  char[100] ;
                    strcpy( this->p , p );
            }
            void change( char *P)
            {       strcpy( this->p, p );
            }
            void show()
            {       printf("\n output is %s" , p );
            }
};
```

```
int  main()
{       String ob1("hello") ;
        String  ob2(ob1);              // ob2 initialized with ob1 by the compiler
        ob1.change("world");           // if ob1 is changed then it also affects to ob2.
        ob1.show();    → World
        ob2.show();    → we expect "hello"  but  shows  "world" → this is wrong output
}
```

As per compiler aspect, when String ob2(ob1) is executed, compiler copies/initializes member to member within the object but not out-side. So it copies only the address in ob1.p to ob2.p, after this, both pointers points to same memory (same string "hello"). As a result, if ob1 is changed then also affects to ob2.
The solution is, we need to write the copy constructor as given below

```
String ob2(ob1);
        ↓
ob2.String(ob1);
        ↓
String ( String  &ob1 )
{       p=new char[100];     // here creating separate memory for ob2.
        strcpy(p, ob1.p);
}
```

Add this constructor to the class, so that compiler calls this while initializing ob2 with ob1 and it creates separate memory for ob2 and then copies the string. This is as shown in assignment overloading concept.

Side effects of copy-constructor

when we overload a copy-constructor to initialize one object with another object like String ob2(ob1). This constructor automatically called too when we pass and return objects between functions. Generally, we don't notice this action. Our copy constructor makes this side effect of un-necessary creation of dynamic memory for parameter objects at the called function. Let us see example

```
int   main()                                  void  set ( String ob )   // String ob=ob1;
{       String  ob1("hello"), ob2;            {       -----------
        ob2.set(ob1);  ——————————————→                -----------
        -----------                                    -----------
}                                             }
```

When ob2.set(ob1) is called in the main() function, ob1 is passed as an argument and initialized to parameter ob. This instruction can be interpreted as "String ob=ob1". Thus our copy-constructor is automatically called by the compiler to initialize ob with ob1. This constructor creates separate memory for ob to store the string. However, creating a separate memory for ob is unnecessary when ob2 string is available.
In this case, for ob, calling the constructor for creating memory, and calling the destructor to delete such memory is useless. Therefore, it is better to avoid calling the copy-constructor & destructor by taking parameter as a reference variable. The above code re-written as

```
int  main()                          void  set( String  &ob )   // String  &ob=ob1;
{       String  ob1("hello");        {       ----------
        ob2.set(ob1);                        ----------
        ----------                   }
}
```

In this code, copy-constructor will not be called, because ob is not an object. (it is a reference)

One more side effect we need to know that, while returning an object from a function, the compiler automatically calls our copy-constructor and creates unnecessary space. Let us see following example

(The following story was already told before; anyway, we are recalling it.)

```
int  add(int a, int b)          int  main()
{                               {       int k;
        return a+b;                     k=add(10,2);
}                                       cout<<k;
                                }
```

Here, the compiler stores the result of a+b value into a temporary variable before returning to main(). Actually, temporary variable is a nameless variable created & destroyed by the compiler. For a while, let us say its name is 'temp'. The data type of 'temp' is depends upon the a+b value-type. This function can be imagined with 'temp' as:

```
int  add( int a, int b )
{       int temp=a+b;
        return temp;
}
```

Now we have learned how the 'temp' variable is created in the program. With this understanding, let us see how objects are returned from a function.

```
String  getCopy()                String  getCopy()
{       -------          →       {       String  temp=*this;     // here temp is created and initialized with *this
        return *this;                    return temp;            // so copy constructor will be called
}                                }
```

The instruction "String temp = *this" does two things. First, it creates memory for temp, and then the copy constructor will be called. When the copy constructor is called, it un-necessarily creates dynamic memory for temp. To avoid this unnecessary creation of 'temp' when while returning, it is better to return references of the object. In this case, the copy constructor will not be called. This can help improve performance and reduce memory usage. The above set() and getCopy() should be re-written as

```
void set( String  &ob )      // String &ob=ob1;
{       p=ob.p;
}

String&  getCopy()
{       return *this;        //  String& temp=*this;
        return temp;
}
```

Let us have one complete example, and notice how many times the constructor/destructor will be called while passing & returning objects between functions.

```
#include<iostream>
#include<string.h>
using namespace std;
class  Test
{       public:
        Test()
        {       cout<<"\n default constructor";
        }
        Test( int v)
        {       cout<<"\n paramerized constructor";
        }
        Test( Test &x )
        {       cout<<"\n copy constructor";
        }
        void set( Test x )          // step2:  copy constructor will be called here  for object “Test X=ob1”
        {
        }                           // step3:  destructor will be called  for  “delete  X”
        Test getCopy()
        {       return *this;       // step4:  copy constructor  will be called here  for  “Test temp=*this”
        }                           // step5:  destructor will be called here  for  “delete  temp”
        ~Test()
        {       cout<<"\n destructor";
        }
};
int main()
{       Test ob1(10), ob2;          //step1:  2 times constructors will be called here for ob1,ob2
        ob2.set(ob1);
        ob2=ob1.getCopy();
}
```

Output

```
parameter constructor  ( for ob1 )
default constructor  ( for ob  2)
copy constructor ( for x )
destructor ( for x)
copy constructor ( for return *this)
destructor  ( for return *this )
destructor  ( for ob1)
destructor  (for ob2)
```

Conclusion

If objects are smaller and does not required overloading a copy-constructor or destructor (like Date and Time class objects), It is often more efficient passing and returning a copy of the objects rather than references. This can improve performance by avoiding implicit pointers for references. In contrast, if objects are larger or objects involved overloading copy constructor with dynamic memory allocation then references are the alternative.

Copy constructor and assignment operator works on same meaning, if we need to overload assignment (=) operator then it is better to overload copy constructor and vice-verse.

Remember: If constructors are involved in dynamic memory allocation, then it is better to write a destructor to delete such dynamic memory; otherwise, memory leaks may occur. (RAM fills with this crashed memory dumps)

Overloading terminal i/o operators << , >>

We can overload input/output operators like any other operators.
Before going to overloading i/o operators (<< , >>) first we understand how they work in C++.
cout<<10; → this prints the integer value 10 on the screen.
cout<<10.45f; → this prints the float value 10.45F on the screen.

We have two API library classes called istream and ostream defined in the iostream.h file.
These are stream classes used for input/output operations in C++.
The cin is an object of istream class, and the cout is an object of ostream class.
These are global-scope objects defined globally in iostream.h file under the std namespace.

The istream class is overloaded with functions for reading all types of input values from the keyboard, while the ostream class is overloaded with functions for writing all types of output values on the screen.
These classes support all data types, including int, float, char, char*, void*, etc.

For each data type, there is a suitable function overloaded in these classes with the operators '<<' and '>>'. The inserter operator **(<<)** inserts values into the output stream (monitor buffer memory), and the inserted values are then reflected on the screen. The extractor operator (>>) extracts values from the input stream(keyboard buffer memory) and assigns them to given variables. These two operators are overloaded in the istream and ostream classes to support i/o operations on all types of data, this is as shown below.

```
int main()
{
    cout<<10;  → cout.operator<<(10);

    cout<<20.50F; → cout.operator<<(20.50F);

    cout<<"hello";  →  cout.operator<<("hello");
}
```

in this way all functions were overloaded in istream & ostream classes.

```
class  ostream
{    -------
     void  operator<< ( int k )
     {   This function prints int value on the screen
         Here it prints 'k' value on the screen
     }
     void  operator<< ( float k )
     {   This function prints float value on the screen
         Here it prints 'k' value on the screen
     }
     void  operator<< ( char *k )
     {   This function prints string value on the screen
         it prints 'k' value on  the screen as string.
     }
     --------
};
```

These inserter operator (<<) and the extractor operator (>>) can be overloaded to print & scan our object values just as built-in types. For example, printing our class-data objects.

```
int  main()
{       Date  ob(10, 5, 2024);
        cout<<ob;
}
```

cout<<ob → The compiler can convert this statement into a function call in two ways.

1) cout.operator << (ob) // it is a member-function
2) operator<< (cout , ob); // it is a non-member-function

The compiler converts based on the function body found in the program, whether it is a member or non-member function. If the function body is written inside the class body, it uses the first style. If the function body is written outside the class body, it uses the second style. It largely depends on you.

In the first style, we would need to write our functions inside the ostream class body, but this is not possible. Since all library classes are supplied in a pre-compiled format, we can't add our own functions to the libraries. I think you got my point: we should write our function in the second style as a non-member.

```
cout<<ob;
   ↓
operator << ( cout , ob );        // here cout is ostream-class object , ob is Date-class object.
   ↓
void  operator << ( ostream  &myCout , Date ob)
{       myCout<<"\n date is :"<< ob.d <<"/"<<ob.m<<"/"<<ob.y;
          or
        cout<<"\n date is :"<<ob.d <<"/"<<ob.m<<"/"<<ob.y;
}
```

Here we can use **cout** or **myCout** to print Date object, since 'myCout' is reference-variable of cout.

Similarly, we can overload the extractor operator(>>) as

```
cin>>ob;
   ↓
operator >> ( cin , ob );      // here cin is  istream-class object ,  ob is Date-class object.
   ↓
void  operator >> ( istream  &myCin , Date  &ob)   // here ob must be reference as to follow the call-by-reference
{       cout<<"enter date:";                        // the input values reflects in the arguments
        myCin>>ob.d>>ob.m>>ob.y;
         or
        cin>>ob.d>>ob.m>>ob.y;
}
```

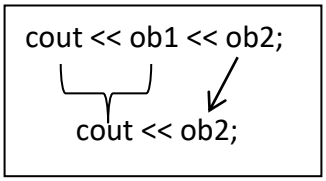
Let us come to full example for reading and writing Date object values through the cin and cout.

```
#include<iostream>
using namespace std;
int  main()
{       Date  ob;
        cin>>ob;            //  operator>> (cin , ob);
        cout<<ob;           //  operator<<(cout , ob);
}
class  Date
{       private:  int  d,m,y;
        public:   friend void  operator<<( ostream &myCout,  Date ob);   // friend to access private data
                  friend void  operator>>( istream &myCin,  Date &ob);
};
void  operator<< ( ostream &myCout,  Date ob)
{       myCout<<"\ndate is :"<< ob.d <<"-"<<ob.m<<"-"<<ob.y;
            OR
        cout<<"\ndate is :"<<ob.d <<"-"<<ob.m<<"-"<<ob.y;
}
void  operator>> ( istream &myCin, Date &ob)      // ob is a reference variable as to follow call-by-reference
{       cout<<"enter date:";                      //  so the input reflects in the arguments
        myCin>>ob.d>>ob.m>>ob.y;
            OR
        cin>>ob.d>>ob.m>>ob.y;
}
```

The operator<<() and operator>>() are declared as friend functions of the class, since they are accessing the private data (d, m, y) of the class.

The above function operaror<<() prints our Date object as mention in the code. However, it can print only a single Date object and not multiple objects like cout<<ob1<<ob2, because after printing ob1, the function does not return the 'cout' object to print the next object ob2. The code should be re-written to support multiple objects as

```
ostream&  operator<< ( ostream  &myCout ,  Date  ob )
{
        myCout<<"\nDate is :"<< ob.d <<"-"<<ob.m<<"-"<<ob.y;
            OR
        cout<<"\nDate is :"<<ob.d <<"-"<<ob.m<<"-"<<ob.y;
        return cout;
}
```



cout<<ob1<<ob2; → here after printing cout<<ob1, the function returns the cout object, allowing us to print ob2 as well. This is as shown in the picture.

Implement the 'Time' class as shown in the main() function, with all input, output, and operator+ functions.

```
int  main()
{       Time ob1, ob2, ob3;
        cin>>ob1>>ob2;
        ob3=ob1+ob2;
        cout<<ob3;
}
class Time
{       private:  int  h, m, s;
        public:   ----------
                  ----------
                  ----------
};
```

Implement the 'Circle' class as shown in the main() function, with all input, output functions.

```
int  main()
{       Circle  ob;
        cin>>ob;
        cout<<ob;   // here calculate and print area value.
}
class  Circle
{       private:  float  radius, area;
        public:   ----------
                  ----------
                  ----------
};
```

string class

In C, there is no data type called 'string' to manage strings like int, float, etc. Where strings are represented using character arrays, and handling them is similar to handling arrays. You may have encountered challenges while handling strings, such as predicting array size, dynamic memory allocation, accessing strings with pointers, pointer leak errors, string assignments, finding substrings, comparing strings, and other related issues.

In C++, the 'string' class is provided to handle strings in a more efficient and convenient way compared to C-style character arrays. The string class abstracts operations like memory management, assignment, comparison, concatenation, substring searching, and more. All these operations are overloaded within the class, making string handling much simpler and more flexible. Here we can handle strings just like int and float types. For example, ob2=ob1 is assignment, ob1+ob2 is concatenation, ob1==ob2 for comparison, etc. Let us have some examples on strings.

C code for concatenation of two strings

```
char a[20]="hello";
char b[20]="world";
char c[20];
strcpy( c , a ) ;
strcat( c , b );
printf(" %s " , c);      // helloworld
```

C++ code for concatenation of two strings

```
string  s1 = "hello";
string  s2 = "world";
string s3;
s3=s1+s2;   // we can add like 2 integers
cout<<s3;
```

C code Comparison of two strings

```
#include<stdio.h>
#include<string.h>
int main()
{   char  a[100], b[100];
    printf("enter string1:");
    scanf("%s", a );
    printf("enter string2:");
    fflush( stdin );
    scanf("%s", b );
    if( strcmp(a,b)==0)  printf("equal");
    else  printf("not equal");
}
```

C++ code Comparision of two strings.

```
#include<iostream>
using namespace std;
int main()
{   string  s1 , s2;
   cout<<"enter string1:";
   cin>>s1;
   cout<<"enter string2:";
   cin>>s2;
   if(s1==s2)    // comparing like 2 integers
   cout<<"equal";
   else  cout<<"not equal";
}
```

Initialization of strings

```
char a[100]="hello world";      // this is c-language string initialization.
string s1="hello world";        // initializing string object with string-literal.
string s2=a;                    // Initializing string object with character array.
string s3=s1;                   // initializing one string object with another string object.
```

Accessing individual char from string

```
string  str="ABCD";
```

str[2] → this gives 3rd character 'C', this is same as accessing array elements.

str.at(2) → this also gives 3rd character 'C', but throws exception if the index is invalid.

Finding length of string using 'string()'

string str = "hello";

str.length() → 5

str.length() or str.size() → 5, both functions are used to find length of string.

length() is specialized for strings, whereas size() works on all libraries like vector, lists, stack, etc.

- If you're working with strings, use length().
- If you're working with other containers like vectors, lists, stack, use size()

Comparison with equal operators(==, !=, <, >, <=, >=)

string st1="hello", str2="world", str3="hello";

str1 == str2 → false, not equal, compares the contents of s1 and s2 (not address)

str1 != str2 → true, equal

str1=="world" → false, not equal

str1==str3 → true, equal

str1 < str2 → true, alphabetical order comparison.

str1 > str2 → false

str1 <= "aaa" → false

"aaa" <= str1 → true

Concatenation of two strings using '+'

string str1, str2, str3;

str1 = "hello"; str2="world";

str3 = str1 + str2; → str3=helloworld

str3 = str1 + " hi"; → str3=hello hi

str3 = "hi " + str1; → str3=hi hello

str3 = "hi" + "world"; → error, at least one operand must be string object.

str3 = str1 + 123; → error, integers cannot be added to string object.

Concatenation of two strings using append() function

string str1="hello" , str2="world";

str1.append(str2); // this is same as str1=str1+str2;

str1.append("bye"); // this is same as str1=str1+"bye";

str1.append("hello", 3); // str1=str1+"hel"; first 3 chars are copied

str1.append(3, 'X'); // str1=str1+"XXX"; 'X' appends 3 times

str1.append(2, 'X'); // str1=str1+"XX"; 'X' append 2 times

str1.append("hello", 2, 3); // str1=str1+"llo"; appends 3 chars from staring index 2

str1.append(str2, 2, 3); // str1=str1+"rld"; appends 3 chars from staring index 2

str3=str1.append(str2); // this is equal to: str3=str1=str1+str2;

---

empty() , clear(), swap(), c_str() functions.

```
string  str1 , str2=”hello”;
```

empty() checks whether string object is empty or not ?

```
if(  str1.empty()==true )
        cout<<”yes”;   → op: yes
else   cout<<”no”;

if(  str2.empty()==true )
        cout<<”yes”;   →  op: no
else   cout<<”no”;
```

clear(): clears the contents of string object

str2.clear() → clears the content “hello” in str2.

swap(): swaps the two string object’s data.

str1.swap(str2) → it swaps the content of str1 and str2.

c_str() → to get a null-terminated C-style string representation.

str2.c_str() → “hello”

String assignments

We can assign strings through assignment operator ‘=’ and also with function assign().

The function assign() overloaded in many forms.

```
string  str1 , str2;
str1=”hello world”;              // assigning string-literal to string object.
str2=str1;                       // assigning one string object with another string object.
str1.assign("hello, world");     // this is same as str1=”hello world”;
str2.assign(str1);               // this is same as str2=str1;
str2.assign(“hello” , 3 );       // assigns only first 3 characters of “hello” to str2.  → str2=”hel”;
~~str2.assign(str1 , 3 );~~      // error, no suitable overloaded function for string.assign(string, int)
                                 // here source str1 must be string literal like “hello”  → string.assign( char*,  int );
str2.assign( str1.c_str() , 3 ); // this is alternative solution for above problem.

str2.assign(“abcdefgh”,3,5 );    // assigns 5 characters starting from index 3,   like str2=”defgh”
~~str2.assign(str1 , 3 , 5 );~~  // no suitable overloaded function available on this.
str2.assign(str1.c_str() , 3 , 5 );  // this is alternative solution for above call statement
str2.assign( 5 , ‘a’ );          // assigns ‘aaaaa’  (repeats 5 times),  like str=”aaaaa”;

~~str2.assign(str1 , 3);~~       // error, here str1 must be string literal as “hello”, that is char* type. (not object)
str2.assign(str1.c_str() , 3);   //  this is alternative solution for above call statement.
```

Searching

str.find(substring) : finds substring in str.

str.find(substring , position): finds the first occurrence of a substring from given position and returns its index

str.find(substring , position, cout): finds the first occurrence of a substring from given position and returns its index it takes 'count' of chars of substring.(not all chars)

string str="ABCDXY";

str.find("XY"); → 4 , "XY" found at 4th index.

str.find("A"); → 0, found at 0th index

str.find('A'); → 0, found at 0th index (can also be used for single char)

str.find("PP"); → -1 , not found,

str.find("AB", 4); → -1 (not found), searches from index 4

str.find("ABCD", 0, 2); → 0 (found), searches for 2 chars "AB" from index 0

str.find("ABCD", 0, 2); → 0 (found), searches for 3 chars "ABC" from index 0

str.find("ABCD", 0, 5); → -1 (not found), searches for 5 chars "ABCDnull" from index 0

str.rfind (substring) → to find the last occurrence of a substring.

str.find_first_of(chars) → to find the first occurrence of any character from a set.

str.find_last_of(chars) → to find the last occurrence of any character from a set.

---

inserting string

object . insert(index , string_literal); // index is position where to inserts given string-literal.

object . insert(index, count , character); // character inserts 'count' of times

object . insert(index, string); // string inserts at given index

object . insert(index, string, from_postion, no.of chars); // n characters are insert from position p of s.

syntax: string_object.insert(index , char or char* or string); // inserts a new string at a specific index position.

syntax1) **str.insert(index , string-literal);** // inserts string-literal in str at index position

```
string str1 = "ABCDEF"
str1.insert( 3 , "XY" );   // inserts XY at 3th index position in str1
cout<<str1; → ABCXYDEF
```

syntax2) **str.insert(index , string-object);** // inserts string-object in str at index position

```
string str1="ABCDEF", str2="XY";
str1.insert( 3 , str2 );   // inserts str2 at 4th index position in str1
cout<<str1; → ABCXYDEF
```

syntax3) string str1 = "ABCDEF", str2="XY";

```
str1.insert(str1.length() , str2);   // inserts XY at end of str1
cout<<str1; → ABCDEFXY
```

syntax4) str.insert(index idx , times t , char c); // inserts 'c' repeatedly 't' times in str at index 'idx'

```
string str1="ABCDEF";
str1.insert( 2, 3, 'X');   // insert 'X' 3 times at 2nd index of str1
cout<< str1; →"hexxxllo"
```

syntax5) str1.insert(index1, str2, index2, count); //reads 'count' of chars from str2 from index2 and writes to str1 at index1.

```
string str1 = "ABCD";
string str2 = "PQRS";
str1.insert(4, str2, 1, 3);     // inserts in st1 from 4th index,  takes 3 chars of str2 from index 1.
cout<<str1;                     //ABCDPQR
```

Deleting substring

string_object**.**erase (index , count); // removes 'count' of characters from index position.

```
string  str="ABCDEFGH";
```

str.erase(2 , 3); → ABFGH , removes 3 characters from index 2

str.erase(); → erases total string

str.erase(3); → ABC , erases all characters from index 3.

str.erase(0); → erases all characters from index 0. (Removes total string)

---

replacing strings

str.replace(index, count, newString) to replace a substring with another.

```
string str="ABCD";
```

str.replace(0, 0 ,"XYZ"); → XYZABCD , replaces 0 chars of ABCD by XYZ at index 0

str.replace(0, 1 ,"XYZ"); → XYZBCD , replaces 1 char of ABCD by XYZ at 0 index

str.replace(0, 2 ,"XYZ"); → XYZCD , replaces 2 chars of ABCD by XYZ at 0 index

str.replace(0, 3 ,"XYZ"); → XYZ , replaces 3 chars of ABCD by XYZ at index 0

str.replace(2, 3 ,"XYZ"); → ABXYZ , replaces 3 chars of ABCD by XYZ at index 2

str.replace(2, 1 ,"XYZ"); → ABXYZD , replaces 1 char of ABCD by XYZ at index 2

Templates

The English word "template" refers to a mould or solid shape that used to create multiple copies from that shape. For example, making several Xerox copies from an original certificate.

In C++, A template defines generic code that can be used with different types of data. For example, a template swap() function can be used to swap integers, floats, characters, etc. Suppose you need to write two swap() functions: one for swapping integers and another for floats. In that case, it is better to write a single template swap() fn that works for both types, since the code (logic) is the same for int and floats.

Thus template allows you to define generic code (logic that works for many types), which can then be used with different data types. Templates are a core feature in C++ that supports **generic programming.**
This eliminates the need to write separate functions or classes for each data type, making the code more flexible and reusable. Templates are used in two ways: **1. Function Template 2. Class template**
syntax for function template is: (here template <> keyword is added on the top of function-body)

```
template < typename  T1 , typename T2, … >        // here template , typename  are keywords
return_type  function_name( parameters_list )     // where T1,T2 are type-name identifiers
{       local_variables_list
        ----------------------
        ----------------------
        return_value;
}
```

Let us create template add() function to add two values.

```
template < typename  T >
T  add( T x  , T  y)
{   T  z;
    z=x+y;
    return z;
}
```

```
void main()
{       int k1;  float k2;
        k1=add(10,20);
        k2=add(10.5f, 20.5f);
        cout<<k1 <<k2;
}
```

Here, **template** and **typename** are keywords, while **T** is the type identifier. For the type identifier **T**, we can use any name, similar to how we name variables. Based on the argument types in function call statements, the compiler generates suitable versions of the function bodies. In this program, the add() function is called with integers and floats, so the compiler generates two versions of the function as overloaded. This concept is to eliminate writing several overloaded functions for the same code. After compilation, the code looks like

```
int  add ( int x  ,  int  y)
{   int  z;
    z=x+y;
    return z;
}
float  add ( float x , float  y)
{   float  z;
    z=x+y;
    return z;
}
```

```
int  main()
{       int k1;  float k2;
        k1=add( 10 , 20 );
        k2=add( 10.5F , 20.5F );
        cout << k1 << k2;
}
```

The following template swap() function is used to swap two values, and it works for types such as int, float, and char. However, it cannot be used to swap two strings because the logic for swapping strings is different from that for int and float. In this case, we need to write an explicit function for strings. It can be taken as function overloading with template function. Let us see following example

```
#include<stdio.h>
#include<string.h>
template<typename T>
void mySwap(T& x, T& y)
{       T z;
        z=x;
        x=y;
        y=z;
}
void   mySwap(char a[],char b[] )   // for strings
{       char c[100];
        strcpy(c,a);
        strcpy(a,b);
        strcpy(b,c);
}
```

```
int main()
    {       int a = 10, b = 20;
            mySwap(a,b);
            printf(" %d   %d", a,b);

            float c=10.5f,d=20.5f;
            mySwap(c,d);
            printf("\n %.2f   %.2f", c,d);

            char e[10]="hello", f[10]="world";
            mySwap(e,f);
            printf("\n %s   %s",e,f);
    }
```

Here, the template generates two versions of the swap() function for swapping integers and float values. Totally three functions will be found in this program as function over-loading.

Template can have more type identifiers where compiler generates suitable function based on calls, for eg.

```
#include<iostream>
using namespace std;
template< typename T1, typename T2 >
void show( T1 x, T2 y )
{
        cout<<"\n output is:"<<x<<"  "<<y;      // we can't use printf() statement here
}
int main()
{       int a = 10, b = 20;
        float c=10.5f,d=20.5f;
        show(a,b);                  // show(int,int)
        show(c,d);                  // show(float,float);
        show(a,c);                  // show(int,float);
        show(c,b);                  // show(float,int);
        show("hello","world");      // show(char*,char*);
}
```

Exercise1: write a template function called sort() , which sorts given N elements of int, float, char and string.
Exercise2: Write a template function called show(), which shows single int/float/char, and array of N values.
Exercise3: Write a template function called reverse(), which reverses int, long int, and string.

class template

Templates can also be applied to classes. Class templates allow you to create a class that can work with any data type. The syntax is almost the same as for function templates; however, when declaring a class object, you need to specify the types explicitly, for example: class-name <int,float>ob1 , <int,char>ob2;
Following template class holds two values, and prints on the screen.

```
#include<iostream>
using namespace std;
template < typename T1 , typename T2 >
class Sample
{       T1 x;   T2 y;
        public:
        void set(T1 x , T2 y)
        {       this->x=x;
                this->y=y;
        }
        void show()
        {       cout<<"\noutput:"<<x<<"  "<<y;
        }
};
```

```
int main()
{       Sample  <int,int> ob1;
        Sample  <float,float> ob2;
        ob1.set(10,20);
        ob2.set(10.5f, 20.5f);
        ob1.show();
        ob2.show();
}
```

Here class object can hold any type of two values; it works for all built-in types. In this example, int and float are stored and printed. Template classes vastly used in data structure's class-libraries like stack, queue, list, tree, graph, set, etc.

Specialized class template

A template class may not work for all data types, as the logic may differ for some types. In such cases, we need to explicitly define the class for that specific type. This is known as a specialized template. For example, the string manipulation may differ from int, float, double, etc.
Following class (classes) holds int, float and string values.

```
#include<iostream>
#include<string.h>
using namespace std;
template<typename T>
class Sample
{  T x;
   public:
   Sample(T x)
   { this->x=x;
   }
   void show()
   {  cout<<"\noutput:"<<x;
   }
};
```

```
// this is specialization
template class for string.

template<>
class Sample<char*>
{  char a[100];
   public:
   Sample(char *p)
   { strcpy(a,p);
   }
   void show()
   {  cout<<"\noutput:"<<a;
   }
};
```

```
int main()
{
    Sample  <int>ob1(10);
    Sample  <float>ob2(10.5f);
    Sample  <char*>ob3("hello");
    ob1.show();
    ob2.show();
    ob3.show();
}
```

Complete the following list template class to hold int,float and strings.

```
template<typename T>
class  List
{   ----------
    ----------
};
int  main()
{       List   <int> ob1;
        List   <float> ob2;
        ob1.insert(10);
        ob1.insert(20);
        ob2.insert(10.5f);
        ob2.insert(20.5f);
        ob1.show();
        ob2.show();
};
```

Exception Handling

An exception is a runtime error that forcibly stops our program during execution. It stops our program in the middle of execution without displaying a proper error message. This unexpected abnormal termination of the program is also referred to as a program crash or program hang.

Errors are typically classified into three types: 1. Syntax errors, 2. Logical errors, and 3. Runtime errors. Syntax errors are handled by the compiler. These errors usually involve missing semicolons, missing quotes, and incorrect use of statements. We don't need to worry about these errors because the compiler catches them at compile time and reports us. Logical errors occur when a program compiles successfully but produces incorrect results. For example, writing avg=m1+m2+m3 instead of avg=(m1+m2+m3)/3. Sometimes, logical errors can also lead to runtime errors. Logical error is also called bug.

Runtime errors stop the program in the middle of execution. Examples include divide by zero, array index out of bounds, file not found, network failure, hardware failure, OS hanging, lack of resources, etc.

Runtime errors also known as "nightmare errors," can create serious issues on the client side. For example, if a banking or railway reservation application crashes during business hours, it can cause significant problems. That is why runtime errors are considered as serious problems in the industry. Many testers are employed to thoroughly test applications in various scenarios before they are delivered to clients.

In C, there were no proper control statements provided to handle exceptions in systematic way. Here exceptions are handled using if-else statement, For example

```
int main()
{       int  X , Y , Z ;
        scanf("%d%d", &X , &Y );
        Z = X / Y;
        printf("output is %d", Z );
}
```

If the input value Y is 0, the computer processor cannot calculate X/0. In such a case, the processor throws back an error to the operating system. The OS then stops our program at this instruction because, without calculating X/Y, the value of Z cannot be printed to the screen. The following code demonstrates how these errors are handled in C using an if-statement.

```
int main()
{       int  X, Y, Z;
        scanf("%d%d", &X, &Y);
        if( Y==0 )
        {       printf("error, denominator is zero");
                return;
        }
        Z=X/Y;
        printf("output is %d", Z);
}
```

Handling various types of exceptions using if-else statements can be complex and limited.
Therefore, C++ provides special control statements to handle exceptions more effectively.
They provided three control statements: **try, catch, and throw.**

Let us see how above program can be written using try-catch statements.

```
int main()
{       int  X, Y, Z;
        scanf("%d%d", &X, &Y);
        try
        {       if( Y==0 )
                    throw Y;
                Z=X/Y;
        }
        catch( int  K )
        {       printf("\n error, the denominator should not be %d ", K);
                return 0;
        }
        printf("output  is %d", Z);
 }
```

Generally, in the try block, input values are examined. If any value is invalid, it is thrown to the catch block using the throw statement, as shown above. The catch block receives the error value and handles it by displaying an appropriate error message or by taking some alternative action. Remember, the try & catch are linked each other (both are pair). In this program, if Y is 0, the throw statement sends the value of Y to the catch block, where it is assigned to K. In the catch block, an error message is displayed along with k value.

A try block can have multiple input validations and throw statements, especially when one validation depends on another. Let us look at a demo program.

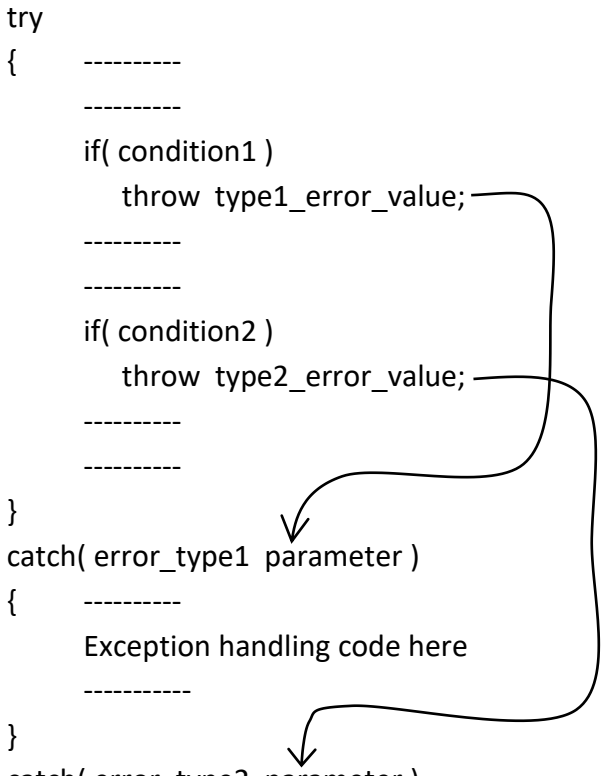
This program accepts time (hours, min, sec) from keyboard and checks whether it is valid or not.

```
#include<stdio.h>
int main()
{       int h, m, s;
        try
        {       printf("\n enter hours :");
                scanf("%d", &h);
                if(h<0 || h>12)
                        throw 1;
                // Control never reaches here if 'hours' is invalid.
                printf("\n enter minutes :");
                scanf("%d", &m);
                if(m<0 || m>59)
                        throw 2;
                // Control never reaches this place if 'minutes' is invalid.
                printf("\n enter seconds :");
                scanf("%d", &s);
                if(s<0 || s>59)
                        throw 3;
        }
```

```
        catch( int v )
        {       if(v==1) printf("invalid hours");
                else if(v==2) printf("invalid minutes");
                else if(v==3) printf("invalid seconds");
                return 0;
        }
        printf("\n u entered time is valid, have a nice day");
        return 0;
}
```

We can have more than one check and throw statement in a try block where one input depends on another. If the hours input is invalid, the program won't scan the minutes. It is unnecessary scanning the minutes when the hours is invalid (so bypassed). Similarly, if the input minutes are invalid, the program won't scan the seconds.

The complete syntax and usage of the try, catch, and throw statements are given below:



```
try
{       ----------
        ----------
        if( condition1 )
            throw  type1_error_value;
        ----------
        ----------
        if( condition2 )
            throw  type2_error_value;
        ----------
        ----------
}
catch( error_type1  parameter )
{       ----------
        Exception handling code here
        -----------
}
catch( error_type2  parameter )
{       -----------
        Exception handling code here
        -----------
}
catch( … )      // three dots, this is the default catch block like in switch statement ( this is optional block also )
{       --------------
        this is the last catch block, catches all errors which are not caught by above catch blocks
}
```

A single try-block can have multiple catch blocks; different catch blocks to handle different types of errors. The error throwing values can be any built-in type or class types. For example

```
try
{       if( a==0) throw 10;
        if( a==1) throw 12.56F;
        if( a==2) throw “hello”;
}
catch( int k )                  // this catch block is used to handle int-type error
{   -----
}
catch( float k )                // this catch block is used to handle float-type error
{   -----
}
catch( … )                      // this catch block to handle other than above type errors
{   -----
}
```

In the try block, errors are examined based on the input values. If any error is detected, the throw statement throws the control to the corresponding catch-block along with the error value/code. The thrown value is caught by the corresponding catch block's parameter, where the exception is handled (resolved).

When an error is thrown in the try block, control immediately jumps to the appropriate catch block, and the remaining bottom statements in the try block are skipped by the compiler. (Observe arrow lines)

You might wonder whether the program continues after the execution of the catch block or stops there. That depends on the instructions written in the catch block. If you use the exit() function, the program terminates at that point. If you use the return statement, the function terminates and returns the control to main() function. If these two are not used, then program continues with the instructions bottom of last catch block. Note that only one catch block will be executed, and the rest will be skipped.

If no catch block matches the error value, the final default catch(…) block will be executed. Here three dots represents and-so-on. This is an optional and default catch-block used to handle unexpected errors which are not caught by the above catch blocks. This must be written as last catch block, if not, it is an error.

If this default-block is not provided and no other catch block suites above, the exception is considered unhandled. In this case compiler calls the implicit function terminate() or abort() to stop the program. These functions terminates program forcibly in the middle of execution.

In C++, for exception handling, there is a built-in mechanism that utilizes the stack data structure, library functions, and API classes. For example, the terminate() function is automatically called when there is no way to move the control after an exception raised.

You may wonder whether such complex systems are necessary to handle simple errors. Certainly not.
In fact, real-time applications often encounter complex scenarios such as nested errors, bypassing errors, handling repeated errors, throwing errors between classes, and more. So, to handle these complex types of errors, this system was provided.

If no errors are found in the try block, control jumps to the bottom instructions after the last catch block, and the program continues smoothly. The execution of try-catch block is somewhat similar to the execution of a switch-case statement. Specifying parameter names at catch-blocks are optional.
The throwing error-values in try-block can be any built-in or class-types. In professional programming, mostly the values are class objects. Let us see demo program throwing exception as a class object

```
#include<iostream>
using namespace std;
class Date
{       int d, m, y;
    public:
        void scan() { … }
        bool  isValid() { … }
        void showError()
        {       cout<<"invalid input date");
        }
};
```

```
void main()
{       Date  ob;
        try
        {       ob.scan();
                if( ob.isValid()==false )
                        throw  ob;
        }
        catch ( Date  p )
        {       p.showError();
                return;
        }
}
```

Following program shows complete picture how try-catch block executes for a given input.

```
#include<stdio.h>
int main()
{       int a; float b;
        scanf("%d%f", &a, &b);
        try
        {       printf("\n reached check point 1");
                if( a<0)  throw a;
                printf("\n reached check point 2");
                if( b==0)  throw b;
                printf("\n reached check point 3");
        }
        catch( int k)
        {       printf("\n int exception raised");
        }
        catch( float k)
        {       printf("\n float exception raised");
                return 0;   // the main() function or program terminates here
        }
        printf("\n reached check point 4");
        printf("\n reached end of program");
}
```

input and output
a=-4 , b=0
reached check point 1
int exception raised
reached check point 4
reached end of program
observe here we did not get
reached check point 2
reached check point 3

input and output
a=9 , b=0
reached check point 1
reached check point 2
float exception raised
observe here we did not get
reached check point 3
reached check point 4

input and output
a=2 , b=2
reached check point 1
reached check point 2
reached check point 3
reached check point 4
reached end of program
observe here we did not get
reached check point 3
reached check point 4

Demo: Scanning month from keyboard and printing total days in that month, if input month not in 1 to 12 then shows an error and again scans until a valid input is entered.

```
#include<stdio.h>
int main()
{       int month;
        int days[]={0,31,28,31,30,31,30,31,31,30,31,30,31};
        while(true)
        {       try
                {       printf("\n\n enter month value:");
                        scanf("%d", &month);
                        if( month<0 || month>12 )
                                throw month;
                        break;
                }
                catch(int m)
                {       printf(" the month should not be %d, try again", m);
                }
        }
        printf(" days in month %d is %d", month, days[month] );
        return 0;
    }
```

```
enter month value:22
the month should not be 22, try again

enter month value:13
the month should not be 13, try again

enter month value:12
days in month 12 is 31
```

Ignoring parameter at catch-block

we can ignore the parameter at catch() block when error value is not important. For example

```
try
{       if( k==0)   throw 10;
        if( k==1)   throw 10.00f;
}
catch( int )      // observe parameter is optional
{       printf("integer exception caught");
}
catch( float )    // observe parameter is optional
{       printf("float exception caught");
}
```

Calling a function inside try-block

The called function can throw an error to the calling function, in this case, the function call statement should be written inside the try-catch block to handle it. If not written inside, then it is said to be unhandled exception and program may crash.

```
#include<stdio.h>
int divide( int a,int b)
{       if(b<=0)  throw b;              //  throwing exception to the caller-function.
        return a/b;
}
int main()
{       try
        {       divide(10,0);           // calling divide() function
        }
        catch( int k )
        {        printf("\n the denominator should not be %d", k);
        }
        printf("\n bye");
 }
```

You might wonder why, instead of throwing an error to the calling function from the called function, why don't we handle such errors within the called function itself. In the industry, functions are written by different people, so it is better to return an exception to him when someone calls a function with invalid arguments. It should be the caller's responsibility to handle such errors when he passes invalid inputs.

For example, if a technical problem is found in a car, the driver reports it to the owner instead of repairing it themselves (not paying from his pockets). In this analogy, the called function is like the driver, and the calling function is like the owner.

Modifying above program, here the call statement has not been written inside the try-catch block, so it leads to program crash if denominator is zero.

```
#include<stdio.h>
int divide( int a,int b)
{       if(b<=0)
                throw b;            //  throwing exception to the caller-function.
        return a/b;
}
int main()
{       divide(10,0);               // observe, this call statement should be written inside try-catch block.
}
```

this program stops abnormally at the instruction a/b, because error is not handling using try-block.

Nested try-catch block

we can write try-catch statement anywhere in the program, even it can be nested like if-statement.
The try-catch can be written inside another try-block or catch-block and also nested any number of times.

nested try in try

```
try
{       try
        {
        }
        catch( )
        {
        }
}
catch()
{
}
```

nested try in catch

```
try
{
}
catch()
{       try
        {
        }
        catch()
        {
        }
}
```

Re-throwing exception

Skipping or bypassing the exception: Using the “throw” statement, we can pass over or skip the exception without handling it. For example, the inner catch block can pass the exception over to the outer catch block. This is called re-throwing an exception from the inner catch block to the outer catch block.

Example1

```
try
{       try
        {   if(k==0)
                throw k;
        }
        catch(int)
        {   printf(“\n caught at inner catch”);
        }
}
catch()
{       printf(“\n caught at outer catch”);
}
```

output: caught at inner catch

Example2

```
try
{       try
        {   if(k==0)
                throw k;
        }
        catch(int)
        {   throw;   // re-throwing to outer block
        }
}
catch()
{       printf(“\n caught at outer catch”);
}
```

output: caught at outer catch

Example3

```
try
{       if(k==0)
            throw 10;
}
catch(int)
{       throw;    // re-throwing  to outside
}
```

This is called unhandled exception, because it is re-throwing to outside and not catching anywhere. So program gets crashed.

Example4

```
try
{       try
        {       if(k==0)  throw 10;
        }
        catch(int)
        {       throw;    // re-throwing  to catch
        }
}
catch(int)
{       throw;    // re-throwing  to outside
}
```

this is also called unhandled exception

Re-throwing exception from called function to calling function

The called function may handle some errors, and some errors can be re-thrown to the calling function using the throw statement. This is extension to the above program.

```
#include<stdio.h>
int test( int p )
{       try
        {       if(p==0) throw 10;
                if(p==1) throw 10.5F;
        }
        catch(int)
        {       printf("\n int EX caught at test() fn");
        }
        catch(float)
        {       throw;      // rethrowing to calling-function
        }
}
```

```
int main()
{       try
        {       test(0);
                test(1);
        }
        catch(int)
        {       printf("\n int EX caught at main() fn ");
        }
        catch(float)
        {       printf("\n float EX caught at main() fn");
        }
        return 0;
}
 output:
    int EX caught at test() fn
    float EX caught at main() fn
```

Restricting some exceptions at called function

some exceptions can be restricted to re-throwing to calling function. In this case, we need to specify list of allowed exception types at function-body declaration. For example: int test(int p) throw(int , float) { … }

```
int main()
{       try
        {       test(0);
                test(3);
        }
        catch(int)
        {       printf("\n int EX caught at main()");
        }
        catch(float)
        {       printf("\n float EX caught at main()");
        }
        catch(char)
        {       printf("\n char EX caught at main()");
        }
        return 0;
}
 output:
    int EX caught at test()
    terminated by char
```

```
#include<stdio.h>
int test( int p ) throw( float , double )
{       try
        {       if(p==0) throw 10;
                if(p==1) throw 10.5f;
                if(p==2) throw 10.5;
                if(p==3) throw 'A';
        }
        catch(int)
        {       printf("\n int EX caught at test()");
        }
        catch(float)
        {       throw;      // re-throwing to calling-function
        }
        catch(double)
        {       throw;      // re-throwing to calling-function
        }
        catch(char)
        {       throw;      // cannot be re-thrown to main() fn
        }                   // only float or double can be
}
```

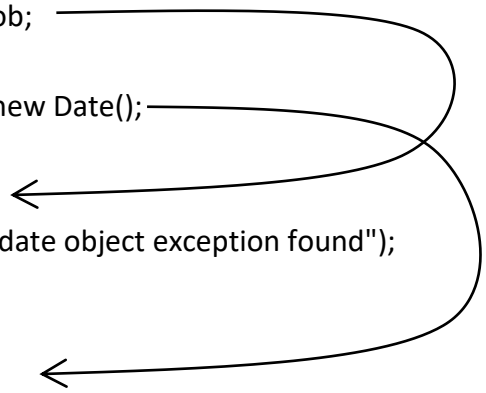
allowed only float, double

this tells that our function is re-throwing only float and double types (not other types)

Throwing class objects

We can throw class object or address of object to the catch-block. Actually, in professional programming objects are thrown and handled. For example

```
#include<stdio.h>
class Date
{
};
int main()
{       Date ob;
        int k=1;
        try
        {       if(k==1)
                    throw ob;
                if(k==2)
                    throw new Date();
        }
        catch( Date ob )
        {       printf("\n date object exception found");
                return 0;
        }
        catch( Date *p )
        {       printf("\n date pointer exception found");
                return 0;
        }
        catch(...)
        {       printf("\n some unknown exception found");
        }
}
```



Note: Exact handling of exception is depends upon client specifications rather than developer ideas.
In real-time, some exceptions are simple like out-of-stocks, network-failure, page-not-found, etc.
However, some exceptions are very critical when many systems are involved in the transactions.
For example, bank payments, train reservation, online order booking & payment, etc. Here a lot of code needs to be written to solve exceptions.

---

guess1:

```
int main()
{       throw 0;
}
```

this is called unhandled exception, so program terminates(crashes)

---

guess2: int main()

```
{       try
        {       if( k==0) throw 10;
                if( k==1) throw 10.5f;
        }
        catch( int )
        {       printf(" int exception raised");
        }
}
```

if k==0 then output is: int exception raised

if k==1 then output is: unhandled exception(crashes)

---

guess3: int main()

```
{       ----------------------------
        try
        {       if( k==0) throw 10;
                if( k==1) throw 10.5f;
        }
        catch( int )
        {       printf(" int exception raised");
        }
        catch( … )
        {       printf(" unknown exception raised");
        }
}
```

if k==0 then output is: int exception raised

if k==1 then output is: unknown exception

---

guess4: int main()

```
{       ----------------------------
        try
        {       if( k==0) throw 10;
                if( k==1) throw 10.5f;
        }
        catch( int )
        {       printf(" int exception raised");
        }
        catch(...)
        {       throw;
        }
}
```

if k==0 then output is: int exception raised

if k==1 then output is: crash (unhandled exception)

The 'throw' statement throws the exception outside of the catch block (passes the exception). Since there is no outer catch block to catch and handle it, this leads to an unhandled exception.

---

guess5:
```
int  main()
{       ----------
        try
        {       if( k==0) throw 10;
                if( k==1) throw 10.5f;
        }
        catch( int )
        {       printf(" int exception raised");
        }
        catch(...)
        {       throw;
        }
}
```
if k==0 then output is: int exception raised

if k==1 then output is: unhandled exception(crashes)

The 'throw' statement throws the exception outside of the catch block (passes the exception). Since there is no outer catch block to catch and handle it, this leads to an unhandled exception.

---

guess6:
```
int test()
{       return 1/0;          // need throw statement to throw an exception
}
int  main()
{       try
        {       test();
        }
        catch( int )
        {       printf("int exception raised");
        }
}
```
this is also unhandled exception.

Inheritance

Inheritance is a core concept of Object-Oriented Programming (OOP) that allows one class to inherit the properties and behaviors of another class. This is similar to how a child inherits assets from their parents, such as cars, homes, or land. The class whose properties are being shared is referred to as the **base class**, while the class that inherits these properties is known as the **derived class**. Nowadays, for easier understanding, the base class is often called the **parent class**, and the derived class is often called the **child class**. When a child class inherits members from a parent class, those members become part of the child class as well. Therefore, all members of the parent class are also available in the child class. In a child object, the members of both the parent and child classes are created side by side.

Let us see the syntax how to derive a class

```
class  X
{   ---------
    ---------
};
class Y  :  access-type  X        //  this inherit access-type can be private/public/protected
{   ---------
    ---------
};
```

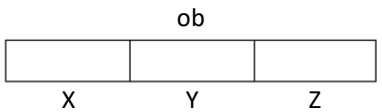
Here, class X is referred to as the base class or parent class, and class Y is referred to as the derived class or child class. class X is independent and does not know about class Y. There is no relationship or dependency of class X on class Y, whereas class Y depends on class X because it inherits its properties.

The inherit access-type for whole class can be private, public, or protected. But most of the time public is used. For a while, we ignore this inherit access-type. Let us see different examples one by one

Demo1:

```
class Parent
{       private: int  X;
         public:  int  Y;
};
class Child : public  Parent
{       public: int Z;

};
int main()
{       Child  ob;
        ----------
        ----------
}
```

This is inherit access type



Since the Child class inherits (derives) all members of the Parent regardless of whether they are public or private, the variables X and Y are inherited by the Child and become part of it. Therefore, space for X and Y is allocated alongside Z in the object ‘ob’. This is as shown above picture how the child class object ‘ob’ occupies in memory.

You might think that private members are strictly private to the class and cannot be inherited. In fact, they are inherited by the child class but are not directly accessible within it. So, space for the private members of the parent is created in the child class object. We know that private data members are accessed through public interface functions from outside the class using set() and get(), in the same way, we can access from the child using these public interface functions. Let us see one by one using examples

```
class Parent
{       private: int  X;
          public: int  Y;
};
class Child : public Parent
{       public: int Z;
        void  show()
        {       printf("%d  %d   %d", X, Y, Z );    //  error, X is private to Parent, not accessible at Child
        }
};
```

The variable X is private to the parent class, so it can't be accessed outside, including by child classes.
In case access is needed, there are three options: the first option is, use public interface functions, specifically set() and get(). This has already been discussed at the beginning of this book. The second option is to change the private access to public access, which would allow access anywhere in the program. However, changing private to public is not a good choice in all cases. (public makes available to all)
The best third option is to change the access type from private to **protected**. Protected works similarly to private but allows access in child classes. Thus, protected is a good choice when members need to be accessed only in the parent and child classes, but not from other parts of the program.
Following example shows how to access private members through public set() and get().

```
class Parent
{       private:   int  X;
          public:  int  Y;
                   int  getX()  {  return X;  }
};
class Child : public Parent
{       public: int Z;
        void  show()
        {       printf("%d  %d   %d ", getX() , Y, Z );     // the function getX() returns the private of X
        }
};
```

Here the getX() function is in public and it returns the X value of Parent class, in this way private are accessed through public interface functions.
Protected works same as private but it allows to access only in parent and child but not in other places.
Actually, in inheritance concept, protected access type commonly used instead of private.
Example for protected access type.

```
class Parent
{       protected:   int  X;
             public:   int  Y;
};
class Child : public Parent
{       public: int Z;
        void  show()
        {       printf("%d  %d   %d", X , Y, Z );    // no error, protected of parent can be accessed in child
        }
};
```

Protected is more convenient access-type which is commonly used in inheritance.

Demo3: Let us have one more example accessing protected members at main() function.

```
class Parent
{       protected:   int  X;
            public:    Parent() { X=100; }
                       void  print()
                       {       printf("%d " ,  X );
                       }
};
class Child : public  Parent
{       protected:   int Y;
            public:    Child() { Y=200; }
                       void show()
                       {       printf(" %d  %d" ,  X, Y );
                       }
};
int  main()
{       Parent  ob1;
        ob1.X=100;          // error, protected members can't be accessed outside of class
        ob1.print();        // calls the print() function of Parent-class
        ob1.show();         // error, there is no show() function in Parent-class

        Child  ob2;
        ob2.Y=200;          // error, protected members can't be accessed outside of class
        ob2.print();        // calls the print() function of Parent-class, as it inherited to the child.
        ob2.show();         // calls the show() of Child-class
}
```

As the child class inherits members from the parent class, these members become part of the child class as well. This means, using an object of the child class, we can access both the child and parent class members in the main() function, provided these members are public. For example, ob2.print() calls the parent class function using the child class object ob2. Note that both classes have their own default constructors. When a child object is created, the parent class constructor is executed first, followed by the child class constructor. (We will discuss about constructors in the next topics.)

Example, accessing public members of parent & child members through child class object at main() function.

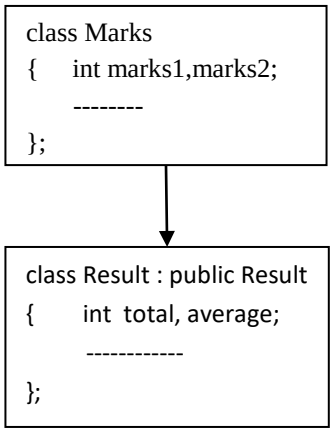
```
class Parent
{   public: int X;
};
class Child: public Parent
{   public: int Y;
            void show()
            {    printf("%d %d", X, Y);
            }
};
```

```
int  main()
{   Child  ob;
    ob.X=100;     // no error, as we can access public from any part
                                                in the program.
    ob.Y=200;
    ob.show();  → 100  200
}
```

As per oop coding, data members should not be accessed outside of classes. This is just a demo program.

Accepting student marks, calculating result and printing on the screen.

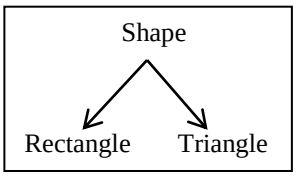
We have written two classes: Marks and Result. The Marks class scans the two subject marks and prints on the screen. The Result class finds the total & average and prints on the screen.




```
#include<stdio.h>
class Marks
{       protected: int marks1 , marks2;
        public:
            void scan()
            {       printf("enter marks of two subjects:");
                    scanf("%d%d", &marks1, &marks2);
            }
            void showMarks()
            {       printf("\n m1=%d,m2=%d", marks1,marks2);
            }
};
class Result: public Marks
{       protected:    int total, average;
        public:
            void find()
            {       total=marks1+marks2;
                    average=total/2;
            }
            void showResult()
            {       printf("\n %d  %d  %d  %d", marks1, marks2, total, average);
            }
            // OR
            void showResult()
            {       showMarks();  or this->showMarks();     // parent class fn called with current object (ob.showMarks)
                    printf(",total=%d,avg=%d", total, average);
            }
};
int main()
{       Result ob;
        ob.scan();
        ob.find();
        ob.showResult();
}
```

Here the function showResult() is calling the parent class function showMarks() to print the marks.
The showResult() is calling with object ob from main() fn, then you may wonder on which object the showMarks() is calling inside the child class. Actually, this is also calling with same object 'ob' through 'this' pointer.

Following example finds area of Rectangle and Triangle.



The Rectangle area is: length*breadth

The Triangle area is: 0.5*length*breadth (this is right-angle triangle)

Here we have taken three classes: 1) Shape 2) Rectangle 3) Triangle.

Shape is the parent class shared by child classes Rectangle and Triangle.

The common properties of child classes are defined in parent 'Shape' so that they can be shared.

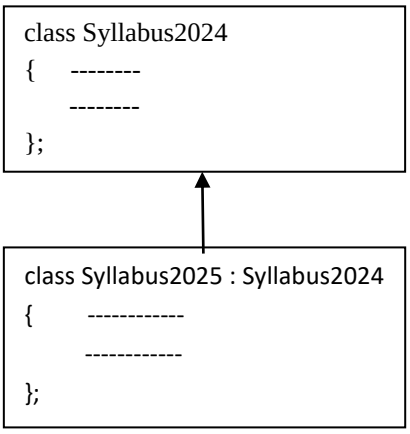
This reduces the repetition of code and increase the flexibility of code. Let us see following code

```
#include<stdio.h>
class Shape
{       protected:  int length, breadth, area;
        public:
        void scan()
        {       printf("enter lenght & breadth:");
                scanf("%d%d", &length, &breadth);
        }
        void show()
        {       printf("\n area is %d", area );
        }
};
class Rectangle: public Shape
{       public:
        void find()
        {       area=length*breadth;
        }
};
class Triangle: public Shape
{       public:
        void find()
        {       area=0.5*length*breadth;
        }
};
int main()
{       Rectangle ob1;
        ob1.scan();
        ob1.find();
        ob1.show();

        Triangle ob2;
        ob2.scan();
        ob2.find();
        ob2.show();
}
```

exercise: Try this program without using inheritance concept (do not use Shape class)

Example extending the code without modifying old code.

In a college, a new subject is added to the syllabus. However, old students follow the old syllabus, while new students follow the new syllabus. This inheritance concept solves the problem by maintaining separate classes for old and new students.

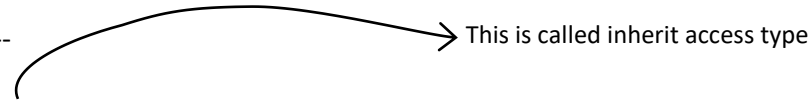



```
#include<stdio.h>
class Syllabus2024
{       protected:   int m1,m2,total,avg;
            public:   void scan()
                      {       printf("\nenter subject1 marks:");
                              scanf("%d",&m1);
                              printf("enter subject2 marks:");
                              scanf("%d",&m2);
                      }
                      void find()
                      {       total=m1+m2;
                              avg=total/2;
                      }
                     void show()
                      {       printf("\n %d %d %d %d",m1,m2,total,avg);
                      }
};
class Syllabus2025 : public Syllabus2024
{       protected:   int m3;
            public:   void scan()
                      {       Syllabus2024::scan();
                              printf("enter subject3 marks:");
                              scanf("%d",&m3);
                              (OR)
                              printf("enter three subject marks:");
                              scanf("%d%d%d",&m1, &m2,&m3);
                      }
                      void find()
                      {       total=m1+m2+m3;
                              avg=total/3;
                     }
                      void show()
                      {       printf("\n %d %d %d %d %d",m1,m2,m3,total,avg);
                      }
};
int main()
{       Syllabus2024    oldObj;
        Syllabus2025   newObj;
        oldObj.scan();   oldObj.find();   oldObj.show();
        newObj.scan();  newObj.find();  newObj.show();
}
```

Inherit access-type

We can specify the access-type while inheriting another class. Let us see following syntax.

```
class  Parent
{       -------------
        -------------
};
class  Child :  public  Parent
{       -------------
        -------------
};
```



The inherit access type can be private or public or protected. But many times public is used by the programmers. Remember default access type is private (if we don't mention any type).

Let us see how this behaves in the program.

Access types while inheriting classes

When inheriting classes, we can specify the access type, which determines how the members are inherited. For example, if a parent has a car and gives it to the child with the condition to use it personally/privately, It means the child can use it freely but cannot give it to others. (Here child cannot give to its child or others)

If a parent class is inherited by a child class using the private access specifier, all protected and public members of the parent are converted to private in the child class. As a result, those protected and public members behave as private in the child class and cannot be accessed by any subsequent child classes. (Access type remains the same at parent class, only conversion is made at the child class)
However, it's important to note that private members are strictly private to that class and cannot be accessed even in the child class (we already discussed this before).

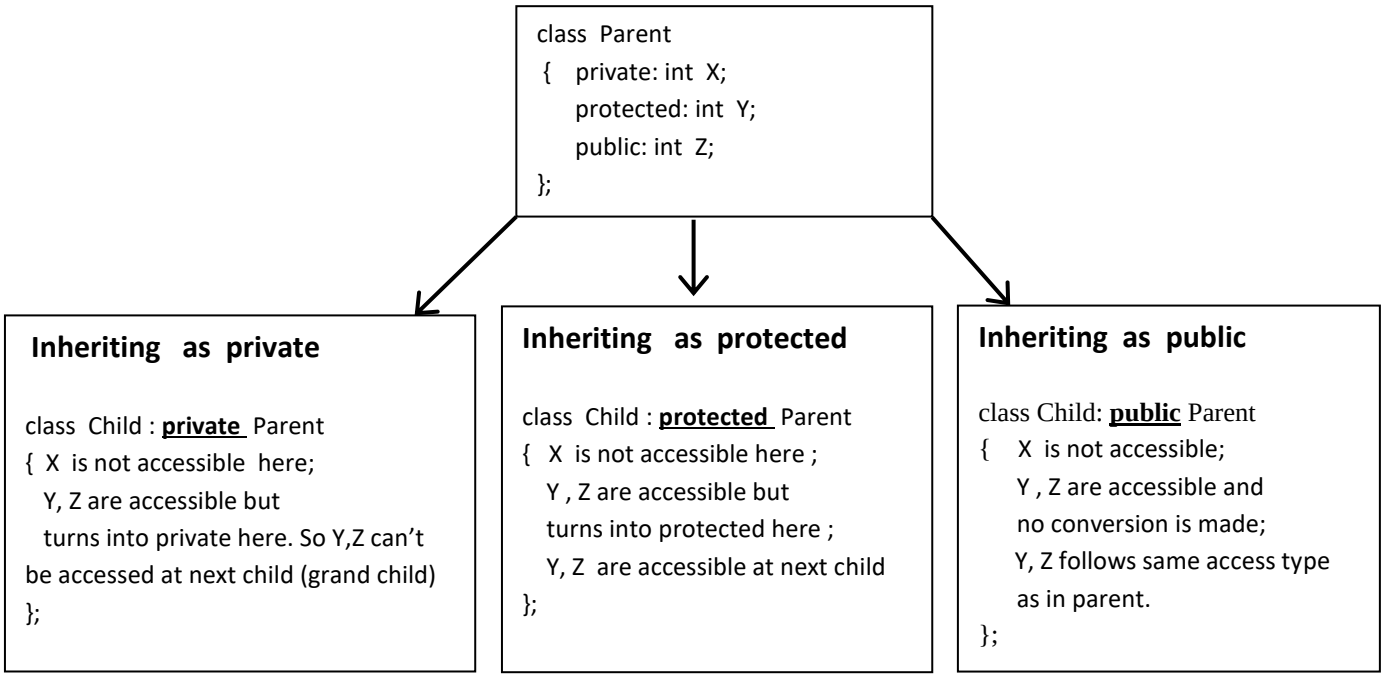
Example with private access type: This demo program explains how parent members are inherited and converted to the given access type in the child class.

```
class Parent
{   private : int  x;
    protected : int  y;
    public :  int  z;                    inheriting as private
};

class Child : private  Parent
{   void show()
    {   x=100;   // error,  private inaccessible
        y=200;   // accessible, and turns into private here
        z=300;   // accessible, and turns into private here
    }
};
```

```
int  main()
{   Parent  p;
    p.x=10;      // error, private not accessible
    p.y=20;      // error, protected not accessible
    p.z=30;      // accessible, because public

    Child  c;
    c.x=10;      // error, private
    c.y=20;      // error, private
    c.z=30;      // error, private
}
```

The object 'p' belongs to Parent class, where p.x is private, p.y is protected, p.z is public;
The object 'c' belongs to Child class, here all c.x , c.y, c.z are private, so they are not accessible outside class.

Following picture explains, how parent members access types are converted at child



Security can be increased but not decreased. This means public members can be converted to protected or private, and protected can be converted to private. However, the reverse conversion is not possible.

Example with protected access type

If access type is protected, it means the child and grandchild (all descendants) can use it freely but not by other classes.

```
class Parent
{   private : int  x;
    protected : int  y;
    public :  int  z;
};

class Child : Protected  Parent
{   void show()
    {   x=100;   // error,  private inaccessible
        y=200;   // accessible, and turns into protected here
        z=300;   // accessible, and turns into protected here
    }
};
```

```
int  main()
{   Parent  p;
    p.x=10;      // error, private not accessible
    p.y=20;      // error, protected not accessible
    p.z=30;      // accessible, public

    Child  c;
    c.x=10;      // error, private not accessible
    c.y=20;      // error, protected not accessible
    c.z=30;      // error, protected not accessible
}
```

The object 'p' belongs to Parent class, where p.x is private, p.y is protected, p.z is public;
The object 'c' belongs to Child class, here c.y, c.z are protected, so they are not accessible outside class

Example with public access type

If access type is public, it means the child can use it freely and child adapts same access type in parent. The child continues to obey the parent's access type.

```
class Parent
{   private : int  x;
    protected : int  y;
    public :  int  z;
};

class Child : public  Parent
{   void show()
    {   x=100;   // error,  private inaccessible
        y=200;   // accessible, remains same access-type
        z=300;   // accessible, remains same access-type
    }
};
```

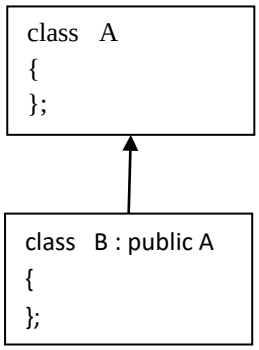
```
int  main()
{   Parent  p ;
    p.x=10;      // error, private not accessible
    p.y=20;      // error, protected not accessible
    p.z=30;      // accessible, public

    Child  c ;
    c.x = 10;    // error, private
    c.y = 20;    // error, protected
    c.z = 30;    // accessible, still public
}
```

The object 'p' belongs to Parent class, where p.x is private, p.y is protected, p.z is public;
The object 'c' belongs to Child class, here c.x, c.y, c.z remains same access type as in parent.

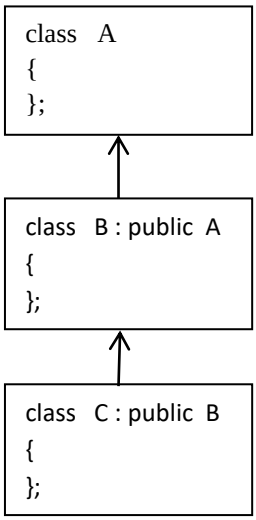
Types of inheritance

in inheritance, classes may inherit in different styles such as one-to-one, one-to-many, many-to-one, and many-to-many. These are mainly categorized into four types: Single-level, Multi-level, Hierarchical, and Multiple (Hybrid)

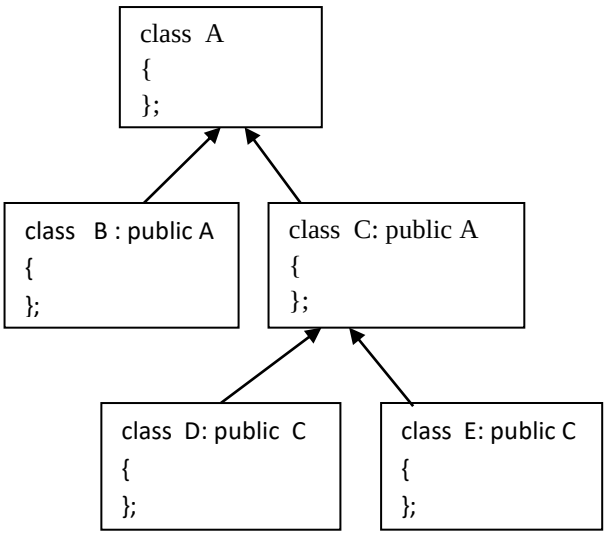
1.Single-level inheritance



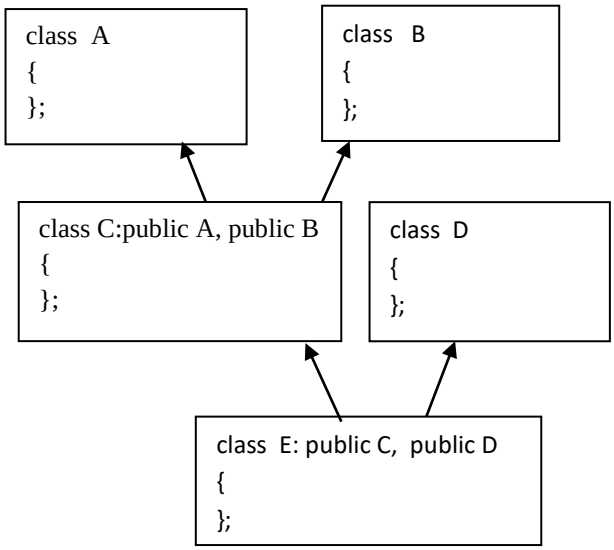
2. Multi-level inheritance



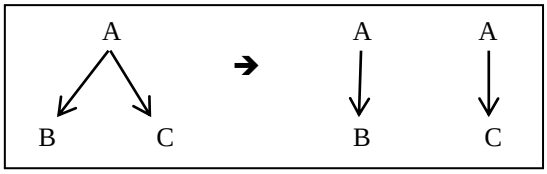
3. Hierarchical inheritance



4. hybrid inheritance

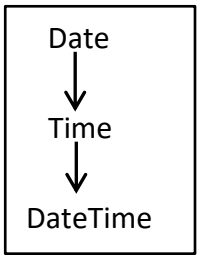


In single-level inheritance, one class inherits from another single class, whereas in multi-level inheritance, more than two classes may be involved one by one. In multiple inheritance, more than two classes are inherited at a time. Hierarchical inheritance is also referred to as single-level or multi-level because, for example, class A is shared by classes B and C. Here, separate copies of A are given to both B and C, and there is no relation between B and C. This is as given below picture.



Example for multi-level inheritance

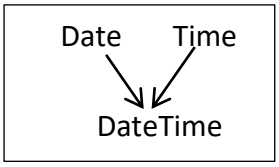
Examples of single-level and hierarchical inheritance models have been introduced earlier.
The following example demonstrates how multi-level and hybrid inheritance are used in a program.
In this example, three classes are created to display the date and time of sample user input values.
This is a Simple example that sets the date and time values to an object and displays them on the screen.
The class hierarchy is: **Date → Time → DateTime → main() fn.**




```
#include<stdio.h>
class Date
{       protected: int day, month, year;
        public:
           void  showDate()
           {      printf("\nDate is %d/%d/%d", day, month, year);
           }
};
class Time: public Date
{       protected: int hour, min, sec;
        public:
             void  showTime()
             {      printf("\nTime is %d:%d:%d", hour, min, sec);
             }
};
class DateTime :  public Time
{       public:
        void  set( int a, int b, int c, int d, int e, int f )
        {       day=a;  month=b;  year=c;
                hour=d;  min=e;  sec=f;
        }
        void  showDateTime()
        {      showDate();
               showTime();
        }
};
int  main()
{       DateTime ob;
        ob.set(31,4,2024,  4,50,26);
        ob.showDateTime();
}
```

Example for multiple/hybrid inheritance

In this example, we use three classes similar to the previous example. However, the Date and Time classes are independent, and both are inherited by the DateTime class, as shown in the diagram. The input and output are the same as in the previous program.



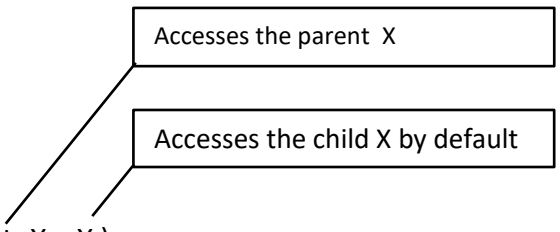

```
#include<stdio.h>
class Date
{       protected: int day, month, year;
        public:
        void  showDate()
        {     printf("\nDate is %d/%d/%d", day, month, year);
        }
};
class Time
{       protected: int hour, min, sec;
        public:
        void  showTime()
        {     printf("\nTime is %d:%d:%d", hour, min, sec);
        }
};
class  DateTime  :  public Date, public Time
{       public:
         void set( int a, int b, int c, int d, int e, int f )
         {      day=a;   month=b;   year=c;
                hour=d;  min=e;   sec=f;
         }
         void  showDateTime()
         {      showDate();
                showTime();
         }
};
int  main()
{        DateTime ob;
         ob.set(31,4,2024,  4,50,26);
        ob.showDateTime();
}
```

shadow/hidden data member

If a data member is declared with the same name in both the parent and child classes, no error occurs; inheritance proceeds as usual. In this case, the parent class member *is* overlapped (shadow/hidden*)* by the child class member. For two members, a separate space is created with the same name at child. When we are in the child class, the child member is accessed by default, and the parent member is hidden. In contrast, when this happens, the compiler always gives preference to local members. In case to access the parent class's member from within the child class, the scope resolution operator is required.

For example

```
class Parent
{       protected:  int x;
};
class Child : public Parent
{       protected:  int x;
         public:   void show()
                   {       printf("%d  %d", Parent::X ,  X );
                   }
};
```



note: This data member over-lapping is called shadowing. Some people also called as over-riding but it is informal. The term overriding is used only for member functions not for data members.

In inheritance, the compiler always first looks for members in the current class. If a member is found, it links to the code. If the member is not found, the compiler searches in the parent, grandparent, and other ancestors, moving upward from the child class to the root class. It always chooses the nearest parent.

For example:

```
#include<stdio.h>
class A
{       public:    int X;
                   A() { X=100; }
};
class B:public A
{       public:    int X;
                   B() { X=200; }
};
class C: public B
{       public: void show()
        {       printf("%d ", X);       // by default class-B version is used. Because it is nearest parent
        }
};
int  main()
{   C   ob;
    ob.show();
    printf("%d ", ob.X );               // by default class-B version is used
    printf("%d ", ob.A::X );
}
```

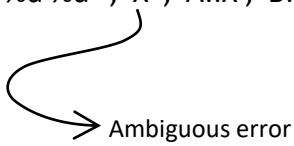
Here the member X is defined in both class-A and class-B, so at class-C, the nearest parent class-B gets accessed. In this way compiler always looks from current class to root class in upward direction.

Shadowing in hybrid inheritance

it is very rare for two base classes to have a member with the same name.
In this case, we need to access explicitly with scope resolution operator(::) otherwise ambiguous error.

```
#include<stdio.h>
class A
{       protected: int X;
        public: A() { X=10; }
};
class B
{       protected: int X;
        public: B() { X=20; }
};
```

```
class C : public A, public B
{       public:
        void show()
        {     printf("%d %d %d ",  X  ,  A::X ,  B::X);
        }
};
int main()
{       C  ob;
        ob.show();  → 10  20
}
```



Here, two variables named X are inherited by the child class C. One X is from class A, and the other is from class B. As a result, the compiler unable to choose which base class's X to be selected, leading to an ambiguity error. The scope resolution operator is required to resolve this ambiguity, as shown in the example above such as A::X and B::X.

If the child class C also has a member named X, by default local member get accessed directly without the scope resolution operator. Let us see following example

```
#include<stdio.h>
class A
{       protected: int X;
};
class B
{       protected: int X;
};
```

```
class C : public A, public B
{       protected:  int  X;
        public:
        void set()
        {   A::X=10;
            B::X=20;
            X=30;     // this  X belongs to C-class
        }
  };
```

overriding member function

Just like data member shadowing, if two functions exist in parent and child classes with the same name, parameter types, and parameter count, then it is considered function overriding. Accessing these functions follows the same approach as described above.

```
#include<stdio.h>
class Parent
{       public:
        void  show()
        {       printf("\n from Parent show() function ");
        }
};
class Child : public Parent
{       public:
        void  show()                    // this child show() function overrides the Parent's show().
        {       printf("\n from Child show() function");
        }
        void display()
        {       Parent::show();         // calls the parent class version
                show();                 // by default calls the child class version
        }
};
int  main()
{       Child  ob;
        ob.show();                      // calls the child version as 'ob' belongs to child-class object
        ob.display();
        ob.Parent::show();              // calls the parent version
}
```

output:

```
from Child show() function
from Parent show() function
from Child show() function
from Parent show() function
```

This is extension to above program: In inheritance, if any method is called with a child class object, the compiler first looks in the child class itself. If it is found, then it links to that function. If it is not found, it searches in the parent, grandparent, and other ancestors, moving upward from the child class to the root class. It always chooses the nearest parent.

```
#include<stdio.h>
class Parent1
{       public:
        void  show()
        {       printf("\n from Parent1 show() function");
        }
};



class Parent2 : public Parent1
```

```
{       public:
        void  show()                    // this child show() function overrides the Parent's show().
        {       printf("\n from Parent2 show() function");
        }
};
class Parent3 : public Parent2
{
};
int  main()
{       Parent3  ob;
        ob.show();                      // calls the parent2 version by default
        ob.Parent1::show();             // calls the parent1 version
}
```

output:

from Parent2 show() function
from Parent1 show() function

Function overriding in hybrid inheritance

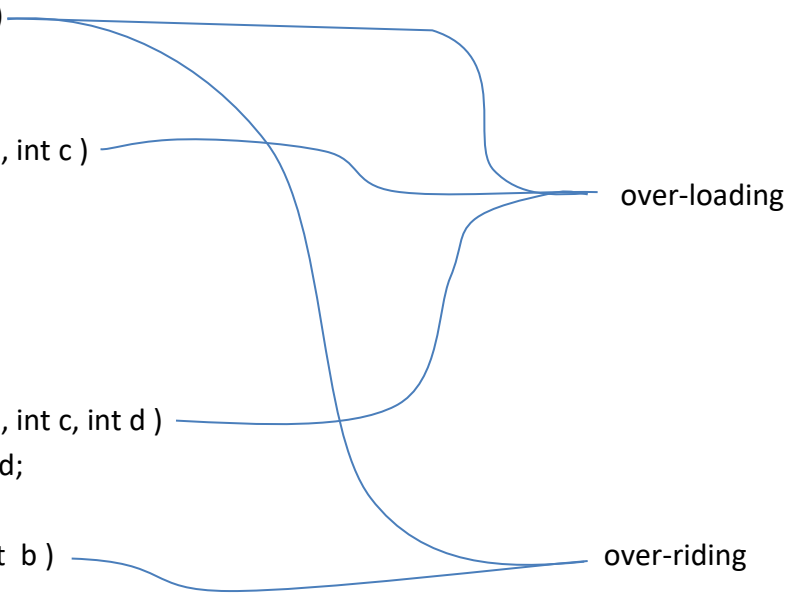
This has already been discussed in the context of data member shadowing; now let us discuss function overriding in multiple inheritance. It is very rare for two base classes to have a member function with the same name. In this case, we need to access explicitly with scope resolution operator(::) otherwise ambiguous error.

```
#include<stdio.h>
class Parent1
{       public:
        void  show()
        {       printf("\n from Parent1 show() function");
        }
};
class Parent2
{       public:
        void  show()                    // this child show() function overrides the Parent's show().
        {       printf("\n from Parent2 show() function");
        }
};
class Parent3 : public Parent1, public Parent2
{       void display()
        {       show();                 // ambiguous error, unable to choose which parent to be selected
                Parent1::show();        // so by using scope resolution, we can solve in this way
                Parent2::show();
        }
};
```

Overloading versus Overriding

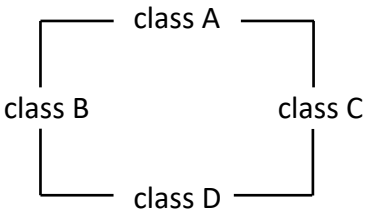
Overloading and overriding can both occur simultaneously in object-oriented programming. In overloading, the parameter list differentiates functions, and overloaded functions do not necessarily belong to the same class—they can sometimes be found in both parent and child classes. Overriding, on the other hand, occurs when a function in the child class has the same prototype as a function in the parent class.

```
class A
{       public:
            int add(int a , int b)
            {     return a+b;
            }
            int  add( int a, int b, int c )
            {     return a+b+c;
            }
};
class B  :  public A
{       public:
            int  add( int a, int b, int c, int d )
            {     return a+b+c+d;
            }
            int    add( int  a , int  b )
            {     return a+b;
            }
};
int    main()
{        B   ob;   int   k;
         k=ob.add(10,20);                // child version will be called
         printf("\n output is %d", k);
         k=ob.add(10,20,30);
         printf("\n output is %d", k);
         k=ob.add(10, 20, 30, 40);
         printf("\n output is %d", k);
}
```



Virtual Base classes

In C++, virtual base classes are used in the context of multiple inheritance to avoid the "diamond problem". When a parent class is inherited two or more times by the child in multiple ways then it leads to diamond problem as shown in this picture



In this case, class A is inherited twice in class D through class B and class C, resulting in two copies of class A in D, which leads to an ambiguity error.

```
#include <iostream>
using namespace std;
class A
{      public:
       void display()
       {     cout <<"hello";
       }
};
class B : public A { };
class C : public A { };
```

```
class D : public B, public C
{
};

int main()
{       D ob;
        ob.display();  // Error: ambiguous call to 'display'
        return 0;
}
```

The solution is to add the virtual keyword when inheriting class A in both class B and class C. Actually, this is a compiler issue — the compiler itself should have been designed to handle this problem. Instead, this responsibility has been passed on to C++ programmers.

```
#include <iostream>
using namespace std;
class A
{   public:
    void display()
    {     cout <<"hello";
    }
};
class B :  virtual  public A { };
class C :  virtual  public A { };
```

```
class D : public B, public C
{
};

int main()
{   D  ob;
    ob.display();      // no error here
    return 0;
}
```

Constructor and Destructors in inheritance

we know, in child class object, memory space is allocated for both parent and child class members.
So when we create an object of the child class, the compiler calls the appropriate constructor of the child class and initializes the members. One might think that this constructor initializes only its own members, not those of the parent class.

Actually, the compiler calls both the parent and child constructors automatically. It executes one by one from the parent to child (top to bottom). Here compiler implicitly calls the parent constructors from its child constructor by adding call statement as shown below. Remember, it calls only the default constructor of parent but not parameterized.

```
#include<iostream>
using namespace std;
class  Parent
{      protected: int  X;
       public:
       Parent()
       {   X=10;
       }
};
class Child :  public Parent
{      protected: int Y;
       public:
       Child()
       {       Y=20;
       }
       Child(int Y)
       {       this->Y=Y;
       }
       void show()
       {      cout<<"\n"<< X<<" "<<Y;
       }
};
int main()
{        Child ob;
         ob.show();
}
output: 10  20
```

```
#include<iostream>
using namespace std;
class  Parent
{      protected: int  X;
       public:
       Parent()
       {   X=10;
       }
};
class Child :  public Parent
{      protected: int Y;
       public:
       Child()
       {      Parent() ;      // this line adds by compiler
              Y=20;
       }
       Child(int Y)
       {      Parent();      // this line adds by compiler
              this->Y=Y;
       }
        void show()
        {      cout<<"\n"<<X<<" "<<Y;
        }
};
int main()
{        Child ob;
         ob.show();
}
```

So, when a child class object is created, control first transfers to the child class constructor, but before executing its body, control is redirected to the parent class constructor. After the parent class finishes, control returns and resumes the child class constructor. So constructors are executed from top to bottom the order how they are derived, whereas destructor are executed from bottom to top in reverse order of constructors.

Remember, by default, compiler calls only the default constructor of the parent as shown above.
To call a parameterized constructor of the parent class, we need to call explicitly with a specific syntax.
Following example explains how parent class parameterized constructor is called from its child class.

Syntax to call parameterized constructor:

```
child-class-name ( parameters list ) : parent-class-name ( arguments list )
{      -----------
       -----------
}
```

```
#include<stdio.h>
  class A
  {    public:  int  X;
               A( int X )
               {    this->X=X;
               }
  };

  class B: public A
  {     public:  int Y;

        B (  int X, int Y ) : A ( X )
        {    this->Y=Y;
        }
        void show()
        {     printf("%d  %d", X, Y );
        }
  };
  int main()
  {     B  ob(10,20);
        ob.show();
  }
```

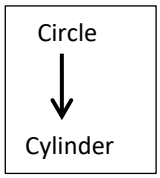
When the instruction B ob(10, 20); is executed, space is created for ob, and then the child class constructor is called. The arguments 10 and 20 are assigned to the parameters X and Y. Here, the value of X is redirected to the parent class constructor. In this way, the constructors are executed

Initializing more than one parent class in multiple inheritance

```
class  Parent1
{   public: int  X;
    Parent1 ( int X )
    {  this->X=X;
    }
};
class  Parent2
{   public: int Y;
    Parent2 ( int Y )
    {  this->Y=Y;
    }
};
```

```
class Child :  public  Parent1 ,  public  Parent2
{  public:  int Z;
   Child ( int X, int Y, int Z ) : Parent1(X) , Parent2(Y)
   {  this->Z=Z;
   }
    void show()
    {     cout<<X<<Y<<Z;
    }
};

int  main()
{    Child  ob(10,20,30);
      ob.show();
}
```

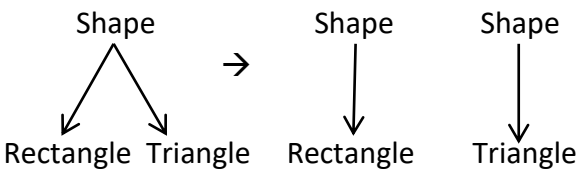
This example explains parameterized constructor of Circle and Cylinder




```
#include<stdio.h>
class Circle
{  protected:  int  radius, area;
   public:
   Circle( int r )
   {  radius=r;
   }
   void printArea()
   {   area=3.14*radius*radius;
       printf("\n area is %f ", area );
   }
};
```

```
class Cylinder : public Circle
{  protected:  int height;
   public:
   Cylinder(int radius, int height) : Circe( radius )
   {  this->height=height;
   }
   void printArea()
   {   area=3.14*radius*radius*height;
       printf("\n area is %f ", area );
   }
};

int  main()
{    Cylinder ob(10 , 4);
     ob.printArea();
}
```

Following example finds area of Rectangle and Triangle using parameterized constructors.




```
#include<stdio.h>
class Shape
{  protected:
      int length, breadth, area;
   public:
      Shape(int l , int  b)
      {  length=l;
         breadth=b;
      }
      void showArea()
      {  printf("\n area is %d", area );
      }
};
```

```
class Rectangle: public Shape
{      public:
       Rectangle(int l,int b): Shape(l,b)
       {
       }
       void findArea()
       {    area=length*breadth;
       }
};
class Triangle: public Shape
{      public:
       Triangle(int l, int u): Shape(l,u)
       {
       }
       void findArea()
       {      area=0.5*length*breadth;
       }
};
int main()
{      Rectangle ob(10,5);
       ob.findArea();
       ob.showArea();
}
```

Destructors in inheritance

Constructors are executed from top to bottom class in the order they are derived in the program.
You might wonder why constructors aren't called from bottom to top instead. Technically, child class members sometimes depend on parent class member's data. For example, the parent class opens a file and the child class writes content to it. Let us take another example, if a parent class dynamically creates the array, and child class writes the data to it. That is why, constructors are executed from top to bottom.

In case of destructors, only one destructor is allowed per class, and these destructors are executed one by one from bottom to top, in reverse order of the constructors.
Following example explains the order how constructers are called in the program.

```
#include<iostream>
using namespace std;
class Parent
{       public:
         Parent()
         {      cout<<"\nParent class Constructor";
         }
        ~Parent()
         {    cout<<"\nParent class Destructor";
         }
 };
 class Child : public Parent
 {       public:
         Child()
         {    cout<<"\nChild class constructor";
         }
        ~Child()
         {    cout<<"\nChild class Destructor";
         }
 };
 int  main()
 {      Child ob;
 }
```

output:
Parent class Constructor
Child class constructor
Child class Destructor
Parent class Destructor

Note: Destructors do not take parameters because there is nothing to do with a dying object except to release already allocated resources, such as files, dynamically allocated memory, and so on.

Example for multiple inheritance, it shows how constructor and destructor are executed in multiple inheritance. It calls same order how they are derived by the child.

```
#include<iostream>
using namespace std;
class Parent1
{       public:
        Parent1()
        {       cout<<"\nParent1 class Constructor";
        }
       ~Parent1()
        {       cout <<"\nParent1 class Destructor";
        }
};

class Parent2
{       public:
         Parent2()
         {      cout<<"\nParent2 class Constructor";
         }
        ~Parent2()
         {      cout<<"\nParent2 class Destructor";
         }
};

class Child : public Parent1, public Parent2
{       public:
          Child()
          {       cout<<"\nChild class constructor";
          }
        ~Child()
          {       cout<<"\nChild class Destructor";
          }
};

int  main()
{        Child ob;
}
```

output:

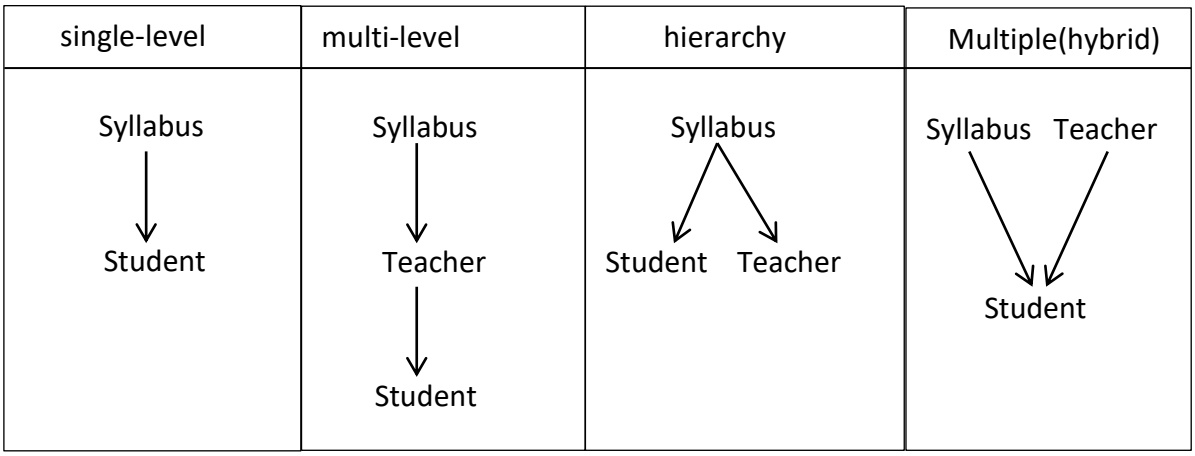
```
Parent1 class Constructor
Parent2 class Constructor
Child class constructor
Child class Destructor
Parent2 class Destructor
Parent1 class Destructor
```

Develop yourself 4 applications to check all types of inheritances. These 4 models specified given below with suitable example and also main() function provided. According to main() function complete classes.

class Syllabus: subject-name (let C++)

class Student: student-name, subject-name (Ram, C++)

class Teacher: teacher-name, subject-name (Sita, C++)



main() function for Single-level

```
int  main()
{     Student ob("Ram", "C++");
      ob.show();  →  student-name: Ram
                     subject-name: C++
}
```

main() function for Multi-level , Hybrid

```
int main()
{     Student  ob("Ram" , "Sita", "C++");
      ob.show();  →  student-name: Ram
                     teacher-name: Sita
                      subject-name: C++
}
```

main() function for Hierarchical (this is same as single-level)

```
int  main()
{     Student ob1("Ram", "C++");
      Teacher ob2("Sita", "C++");
      ob1.show();  →  student-name: Ram
                      subject-name: C++
      ob2.show();  →  student-name: Sita
                      subject-name: C++
}
```

Virtual functions

a virtual function is a function declared in the parent class, either with or without a body, and redefined in the child class. Here, a parent class pointer can points to any child class object, and based on the object the pointer is pointing to, the virtual function of that class is called.

To make a function virtual, add the keyword 'virtual' before the function name in the parent class.

Let us see following example

```
#include<iostream>
using namespace std;
class Vehicle
{       public:
                virtual int  getPrice()
                {   return 0;
                }
};
class Bike : public  Vehicle
{       public:
                int getPrice()
                {    return 75000;
                }
};
class Car : public  Vehicle
{       public:
                virtual int getPrice()
                {   return 800000;
                }
};
int  main()
{       Vehicle  *ptr;  int choice;
        cout<<"enter choice (1 for bike, 2 for car ) : ";
        cin>>choice;
        if( choice==1)  ptr=new Bike();
        else   ptr=new Car();
        cout<< ptr->getPrice();          // this calls the Bike/Car class version based on the input value.
}
```

Here, the pointer ptr is of the parent class Vehicle type, but it is assigned an object of the child class Bike or Car. When we call ptr->getPrice(), the child class version of getPrice() is executed because it is declared as a virtual function.

If getPrice() is not declared as virtual, then the parent class version will be called instead of the child's.
In this case, the function is called based on the pointer type, not on the object type it is pointing to.

Virtual functions ensure that the correct version is called based on the type of the object pointed to by the pointer. When we use the keyword virtual, the virtual mechanism is enabled. To support this feature, the compiler implicitly maintains a table called the V-Table, which stores the addresses of all virtual functions from the parent to the child. At runtime, function calls are resolved by searching this table based on the actual type of the object pointed to by the pointer. The concept of virtual functions comes under run-time polymorphism.

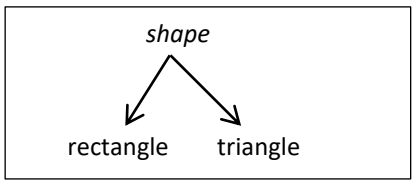
In C++, polymorphism is achieved in two ways: compile-time and runtime.

Function and operator overloading are examples of compile-time polymorphism, while virtual functions are examples of run-time polymorphism. In compile-time polymorphism, the function call is linked to the function body during compilation. That's why it's also called static or early binding.

In run-time polymorphism, the function call is linked to the function body while the program is running, depending on the input values. This is also known as dynamic or late binding, and it is achieved using virtual functions as said above.

Early binding is fast and efficient but lacks flexibility. Late binding, on the other hand, is slower because functions are selected based on input values, but it provides greater flexibility, which is beneficial for large projects.

Let us take one more practical example: calculating area of rectangle or triangle from a given choice of input.




```
#include<iostream>
using namespace std;
class Shape
{       protected: float  length, breadth, area;
        public:
        virtual void  findArea()
        {   area=0;
        }
        void showArea()
        {   cout<<"area is :"<< area;
        }
};
```

```
class Rectangle : public Shape
{       public:
        Rectangle(int l , int b)
        {   length=l; breadth=b;
        }
        void findArea()
        {   area=length * breadth;
        }
};
```

```
class Triangle : public Shape
{       public:
        Triangle(int l , int b)
        {   length=l; breadth=b;
        }
        void findArea()
        {   area=0.5F * length * breadth;
        }
};
```

```
int main()
{       Shape *ptr;     int  choice;
        cout<<"enter choice 1 or 2:";          // 1 for rectangle, 2 for triangle
        cin>>choice;
        if( choice==1 )
                ptr=new Rectangle(3,4);
        else
                ptr=new Triangle(3,4);
        ptr->findArea();
        ptr->showArea();
}
```

Here, ptr can points to either Rectangle or Triangle based on user 'choice' input. Depending on the type of object that ptr points to, the appropriate class version of the findArea() function will be called.
Note1: Adding the keyword virtual to the function in the child class is optional.
Note2: If findArea() is not overridden in child class, then parent version will be called.

Note: The concept of virtual functions (runtime polymorphism) applies only to functions declared as virtual, not to all functions of the class. If a function is not virtual, the compiler does not maintain its address in the V-Table, where it handled as normal inheritance. Let us following example

```
#include<iostream>
using namespace std;
class  Parent
{       public:
        virtual  void show()
        {       cout<<"\n Parent class virtual show() function called ";
        }
        void  display()
        {       cout<<"\n Parent class non-virtual display() function called ";
        }
};
class Child : public Parent
{       public:
        void  show()
        {       cout<<"\n Child class  virtual show() function called ";
        }
        void  display()
        {       cout<<"\n Child class  non-virtual display() function called ";
        }
};
int   main()
{       Parent  *ptr;   Child ob;
        ptr=&ob;
        ptr->show();        → Child class virtual show() function called
        ptr->display();     → Parent class non-virtual display() function called ( not child class version )
}
```

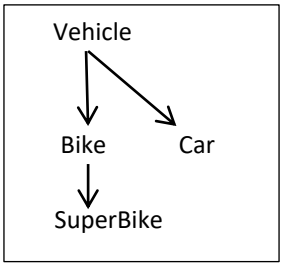
The instruction ptr->show() calls the Child version instead of the parent version. Since this function is virtual. In contrast, the instruction ptr->display() calls the Parent version instead of the Child version because display() is a normal member function, and the virtual mechanism is not enabled for this function.
In this case, the compiler calls the function based on the type of ptr rather than the type of the object it points to.

---

virtual functions in multi-level inheritance

In multi-level inheritance, the virtual function can be overridden by all child classes relative to its type. Both the child and grandchild can have their own overridden functions. Don't worry; the compiler always selects the appropriate class based on the object type. If the virtual function is not overridden in the child class, the compiler searches upward direction through the parent, grandparent, and great-grandparent (ancestor classes) and calls the nearest ancestor class version.

Let us have some examples




```
#include<stdio.h>
class  Vehicle
{       public:
        virtual  void  showPrice()
        {       printf("\n cost not defined");
        }
};
class  Bike : public Vehicle
{       public:
        virtual void  showPrice()
        {       printf("\n cost around 1 to 2 lakhs");
        }
};
class  SuperBike : public Bike
{
};
class  Car :  public Vehicle
{
};
int  main()
{       Vehicle   *ptr;
        ptr=new Bike();
        ptr->showPrice();          // cost around 1 to 2 lakhs, calls Bike class version

        ptr=new  SuperBike();
        ptr->showPrice();          // cost around 1 to 2 lakhs, calls Bike class version

        ptr=new Car();
        ptr->showPrice();          // cost not defined, calls Vehicle class version
}
```

It is important to note that, using a base class pointer, we can call methods of the parent class as well as the overridden methods of the child class, but not the individual methods of the child class.

Let us consider the following example

```
#include<stdio.h>
class Vehicle
{
};
class Bike : public Vehicle
{       public:
        void showPrice()
        {       printf("75000”);
        }
};
int  main()
{       Vehicle  *ptr;   Bike  ob;
        ptr=&ob ;
        ptr->show();   // error showPrice() not found in Vehicle
        ob.show();      // no error
}
```

For dynamic polymorphism, the base class always defines all the interface functions required by the derived classes. These interface functions are nothing but virtual functions. These functions are essential in API programming. If any individual functions are written in the derived classes, they are used by these interface functions to support the code. These functions assist the interface functions in coding.

Pure virtual function

a virtual function can be defined in two ways: either with or without a body. When defined with a body, it is considered a normal virtual function. The functions provided in all examples above are normal virtual functions. When defined without a body, it is considered a pure virtual function. A pure virtual function is assigned a value of zero. For example:

```
virtual void showPrice()=0;
```

This declaration indicates that, it is a pure virtual function. The purpose of a pure virtual function is to **force** the programmer to override the function in the child class. If not overridden, it results in a compile-time error.

```
class  Vehicle
{   virtual  void   showPrice()=0;      // this is pure virtual function.
};
class  Bike : public Vehicle
{
};
int  main()
{       Vehicle *ptr=new Bike();
        ptr->showPrice();               // compiler shows error at this statement
}
```

Here the function call ptr->showPrice() gives compile error. It must be overridden in child class.

Pure virtual function and abstract class

a pure virtual function is considered incomplete function because it has no function body.
It is assigned a value of zero. For example: **void showPrice()=0;** This declaration specifies only the function name, parameter list, and return type, essentially serving as a function prototype.

This declaration enforces that the programmer must override the function in the child class. Thus, making a virtual function pure is a way to guarantees that a derived class will provide its own redefinition.

If a class contains a pure virtual function, it is considered an incomplete class. A pure virtual function is treated as an incomplete function because it does not have a function body. This incomplete function makes the class incomplete as well. Such a class is referred to as an abstract class and we cannot create an object of incomplete class. For example

```
class  Vehicle                      // this is called abstract class
{      virtual void  showPrice()=0;
};

int  main()
{      Vehicle ob;                  //  here error , we can create object of abstract class
       Vehicle   *ptr;              //  here no error, we can take pointer of abstract class.
}
```

So we cannot create an object of an abstract class because it is incomplete. Even if virtual is not implemented in child class, then child also becomes as abstract class.

We can have a pointer of an abstract class because it can point to a fully implemented child class object. Therefore, it is allowed to have a pointer to an abstract class.

The following example explains a normal virtual function. This function may or may not be overridden in the child class. If it is not overridden, the parent class version will be called. Here we can create an object of this parent class.

```
class  Vehicle                      // this is complete or concrete class
{      virtual void  showPrice();
       {       cout<<"price not defined";
       }
};
class Bike : public Vehicle
{
};
int  main()
{      Bike  ob;                    // no error, we  can create object of this class.
       Vehicle   *ptr;
       ptr=new Bike();
       ptr->showPrice();            // no error, parent version will be called.
       Vehicle  ob2;
       ob2.showPrice();             // no error
}
```

Virtual Destructor

when a child class object is created, the constructors are executed from the parent class down to the child class, while destructors are executed in the reverse order. However, in the case of virtual functions, when using a parent class pointer to point to a child object, the compiler only calls the parent class destructor. This happens because the destructor is called based on the pointer type rather than the type of the object being pointed to. The solution is to make the base class destructor virtual. By doing so, the compiler calls all destructors in the inheritance chain.

Destructor without virtual

```
#include<stdio.h>
class A
{       public:
        ~A()
         {      printf("\nparent destructor called");
         }
};
class B : public A
{       public:
        ~B()
        {       printf("\n child destructor called");
        }
};
int main()
{        A    *ptr;
         ptr=new B();
         delete ptr;
}
```

output:

```
parent destructor called
```

Destructor with virtual

```
#include<stdio.h>
class A
{       public:
        virtual ~A()
        {       printf("\nparent destructor called");
        }
};
class B : public A
{       public:
        ~B()
         {      printf("\nchild destructor called");
         }
};
int main()
{       A   *ptr;
        ptr=new B();
        delete ptr;
}
```

output:

```
child destructor called
parent destructor called
```

Conclusion

Polymorphism is the process by which a common interface is applied to two or more similar (but technically different) situations, this implementing the “one interface multiple methods” philosophy. Polymorphism is important because, it can greatly simplify complex systems. A single, well-defined interface is used to access a number of different but related actions, and artificial complexity is removed. In essence, polymorphism allows the program is easier to understand and maintain. When related actions are accessed through a common interface, you have less to remember.

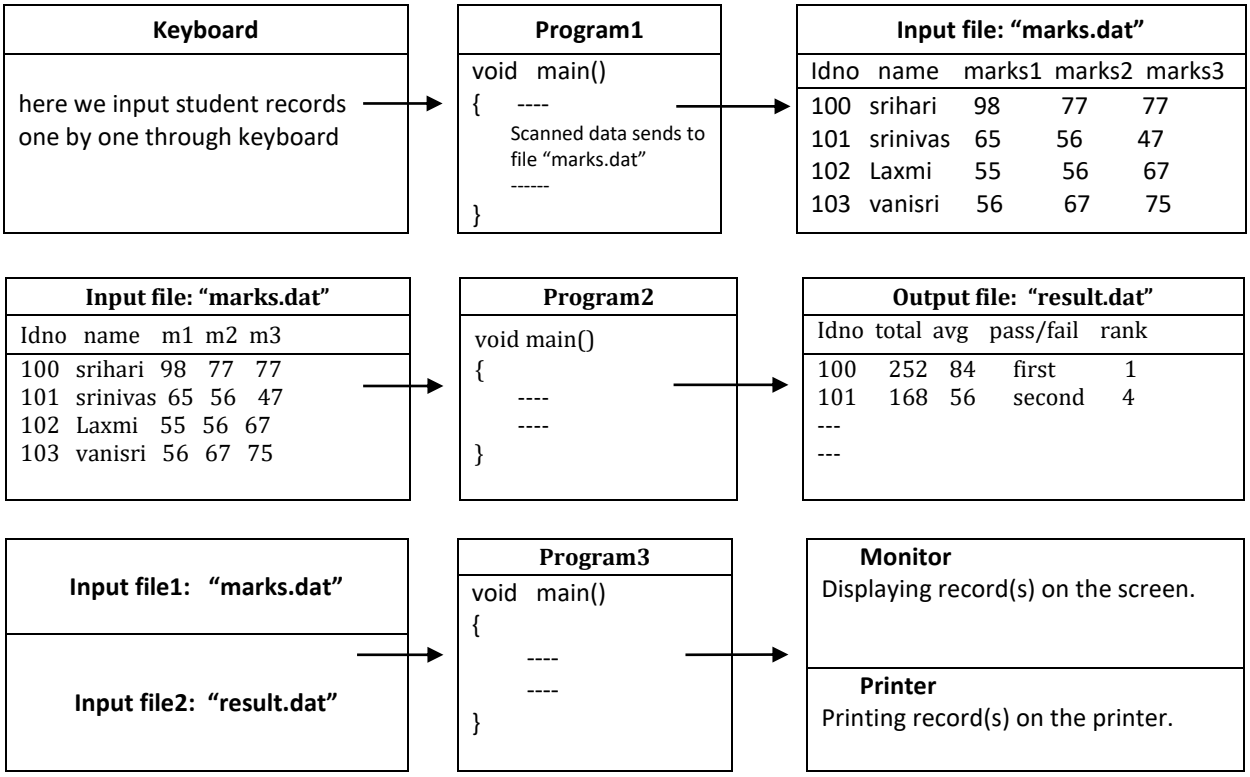
In C++, polymorphism is achieved in two ways: compile-time binding and runtime binding. Function overloading and operator overloading fall under compile-time binding, while virtual functions fall under runtime binding. Early binding is fast and efficient but lacks flexibility. Late binding, on the other hand, is slower because functions are selected based on input values, but it provides greater flexibility, which is beneficial for large projects.

File handling in C++

In real life applications, some kind of data needs to be accessed more times in present and future purpose like bank transactions, business transactions, etc. Therefore such data is kept permanently in the computer, so that it can be accessed many times later.

For this purpose, secondary storage devices are used to store the data permanently in the form of files. These files often called data files. Thus, data file can be defined as collection of data, stored permanently in secondary storage device. The secondary storage devices such as hard-disk, cd, dvd, pendrive, tape, etc. Each file is identified by a valid name called **file-name**. The interaction with files is treated as interaction with io devices in the computers. Therefore, all secondary memory devices treated as **io** devices.

Let us consider a menu driven program to automate student marks in a college. Let the marks scanned from keyboard are stored permanently in a data file called "marks.dat", later, such marks are processed and stored the result in a separate file called "result.dat" or displayed on the screen. This is depending upon the requirement of problem. Following picture explains how our programs interact with files.



In computer science, files are classified into two types, 1. Data files, 2. Executable files.

1. Data files: for example, pdf files, jpg image files, mp3 sound files, mp4 video files, business data files, and program's source code files like c, cpp, java files, etc.

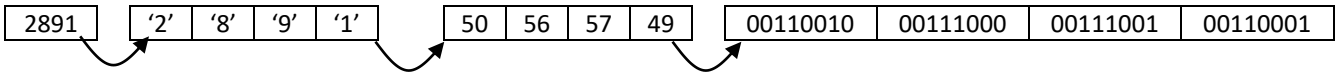
2. Executable program files: for example, **.**exe, .com, .dll, .class, .obj, etc; Contains machine code instructions. Executable files are created using compilers, whereas data files are created using executable files. In computer, any file is stored either in **text & binary** format. All executable files are stored in binary format, whereas data files are stored in text or binary formats depending on our requirement.

1. Text Format Files: here, the data is stored in the form of ascii values (string format)

2. Binary Format Files: here, the data is stored as it is in program variables (raw format)

Text Files

In text files, everything is stored in the form of ascii values, even **numeric data** is converted into string format with its ascii values of each digit and stored in a file. For example, **'k'** contained a value 2891, this number is stored in text file as



Here 50, 56, 57,49 are ascii values of 2, 8, 9,1 respectively. Internally each digit ascii code is again stored in binary form as shown above (as computer understands only binary values), in this way, the text data is stored and retrieved from text files. The library functions converts this numeric to text and text to numeric while storing and reading back from files. These text files are only suitable for public sharing related files such as document files, help files, message files, source code program files, small data files, configuration files, etc. In general, these files are handled in two ways in the computer, by writing a special software program to manage them, or by using ready-made text editor softwares. Using text editors, we can read/write/modify the text data. We have several text editors like notepad, word pad, ms-word, etc.

In text files, the '\n' character is stored two characters as "\r\n"(unix format), but while reading back, it reads as single character (\n). The library functions does this job while writing-reading '\n' to files, so we don't need to worry about how to read and write this '\n' character to a file.

In text files, the value 26 is inserted as end-of-file mark (this is like null-char '\0' for strings). Here some people raise a doubt, if file data itself contained 26 like employee age, then how it differentiates with the end-of-file 26. Our 26 is stored as '2' and '6' as its ascii values 50 & 54, so no conflict between these two.

Binary files

In this kind of file organization, the data is stored as it is in program variables. Here no conversion is made while storing and reading back unlike text files. Generally, binary files are extensively used for maintaining similar type of data like employee records or bank accounts, insurance accounts, etc.
In binary i/o, as it is of storing & reading back takes place between the RAM & hard disk, therefore binary files are faster than text files. At the end of text files, the value 26 is inserted as end-of-file-mark, whereas in binary files no such value is inserted, where end-of-file mark is known by the file-size.

In case of binary files, we cannot manipulate the data using text editors. For example, take employee structure: idno | name | age → int | char(30) | int . If this structure data is dumped as it is into file, then it has to be read back as it is. (2byte at a time for idno, 30byte at a time for name, and 2 byte for age)
So, text editors unaware of this user-defined structure format, hence we cannot handle binary files using text-editors.(text editors are designed to read/write the data only in the form of ascii values of byte by byte)

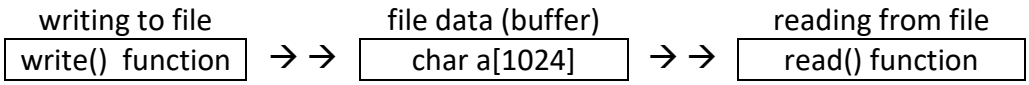
There should be special software program needed to handle binary files. For example 'pdf' file is a binary format file and cannot be opened in text editor, a special software like ***Adobe-R*eader** is used. Similarly, jpg files, image files, sound files, movie files are managed using special software.

File Streaming or File Buffering

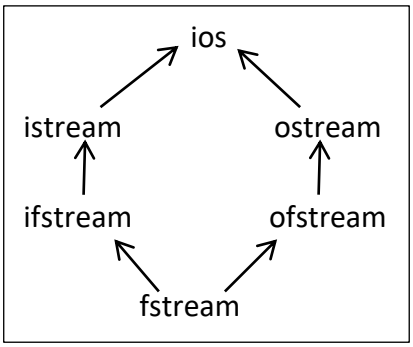
The word 'stream' defines a continuous flow without interruption. For example, in a household, the water tap is connected to the head water tank rather than directly to the motor pump. The motor pump fills the head water tank, and water is drawn through the pipeline. This setup provides hassle-free access to water at any time, ensuring a continuous flow without interruption. This system can therefore be referred to as water streaming. The other example is, video streaming, the youtube channel uses this technique to play videos seamlessly.

Let us come to file streaming: The total hard disk space is divided into several blocks, where each block size 1024/2048 bytes. This is not fixed size and it may vary. Our file content is divided according to this hard disk block size and stored in the hard disk. If file size is 3000 bytes then it takes two or more blocks. We can't do any insert/delete/modify operations on file data when file is in hard disk, because, the hard disk supports only block by block reading-writing. So first we dump the file data from hard disk to RAM and then we do respective operations, later we store back updated data to disk. RAM supports byte-by-byte read-write operations therefore we can do any operations when data is in RAM.

When we open an existing file, the ***total file*** contents would not be dumped into RAM, only the first block dumped into RAM and then file pointer set to it, the remaining blocks are loaded one by one when file pointer moves advance. File pointer is a pointer used to read/write file contents. When a file pointer moves end of first block, the second block loads into same memory (1st block replaces by 2nd block), in this way files data loads into RAM and operated, this process is called file-streaming and managed with help of operating system. For random file organizations, the blocks are loaded & replaced randomly as per movement of file pointer. OS provides this file streaming facility to move the file-data from disk-to-RAM and RAM-to-disk. Our C++ API classes interact with OS, we don't need to bother about how to manage this file-streaming.
Stream i/o → array of bytes memory + read() fn + write() fn + open() fn + close() fn + other functions.
This array of bytes memory is also called buffer or stream memory with all functionality.



In C++ for handling file stream, there are 3 predefined classes with plenty of functions available.
ifstream class: this is for Input file stream, used for reading data from files.
ofstream class: this is for output file stream, used for writing data to files.
fstream class: this is for both input and output file stream, used for both i/o operations on same file.
These 3 classes are defined in a header file called "fstream.h".
So we need to include this file in our program as: #include <fstream> or #include <fstream.h>
ifstream class is derived from istream, and this istream class is again derived from ios class.
ofstream class is derived from ostream, and this ostream class is again derived from ios class.
Let us see class hierarchy



We have already learned about classes istream and ostream, these are used for console i/o to read-write values from keyboard-screen using cin/cout. These classes have various functions to convert numeric data to text and text data to numeric. That is why these classes are inherited by ifsream and ofstream.
The ios class has various flags which are used while opening files.

ios::in → read-mode, indicates opening a file in read-mode
ios::out → write-mode, indicates opening a file in write mode
ios::binary → binary-mode, indicates opening a file in binary mode
ios::app → append-mode, indicates opening a file in append mode, we cannot modify old data.
ios::ate → at end, by opening a file, the pointer moves end, but also allows to modify old data.
ios::trunc → truncate(removes) old content, and new fresh data is added.
eof() → to know file pointer is moved to end of file or not
tellg() → it tells the current position of read/get pointer in a file
tellp() → it tells the current position of write/put pointer in a file
seekg() → moves the read/get pointer to a specified position
seekp() → moves the write/put pointer to a specified position
is_open() → to know file is successfully opened or not, we can also use not operator (!)
We can do bitwise OR operator on these flags. For example, "ios::in|ios::out" allows both read-write operations. The flags "ios::in|ios::binary" opens a file for reading with binary mode.

Operations on files are mainly three

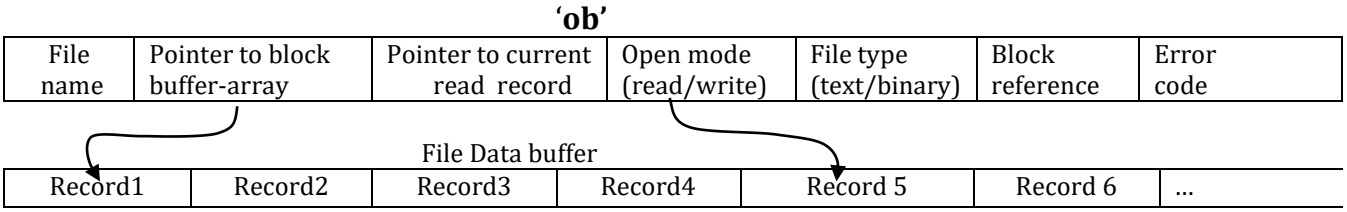
1. Opening a file
2. Applying read/write/modify operations on file data
3. Closing a file

Opening a file in read mode

The function open() is used to open a file, when we open a file, the file contents are dumped from hard-disk to RAM and then file object set to it. Let us see how to use the function,

```
int main()
{       ifstream ob;                    // file object for 'ob' reading
        ob.open("input.dat", io::in );  // here "input.dat" is name of file which we want to open, ios :: in is optional
        If ( ! ob )                     // ob.operator!() → to check file is opened or not
        {       cout<< "file not found or not opened";
                return;
        }
        -------
}
```

with this file object ob, we can do only read operations because it belongs to ifstream class. The operator not [!ob → ob.operator!()] is overloaded in this class to check whether the file is opened or not. This operator function returns bool value. When a file is opened in the RAM then file object 'ob' holds the file data as given below picture.



The function open() fills the file information in the object 'ob' as above shown. The buffer-array, buffer-size(block-size), pointer to buffer, pointer to current read/write byte, etc.

Opening a file in write mode

```
int  main()
{       ofstream  ob;                       // file object 'ob' for writing
        ob.open("output.dat" , ios::out);   // here "output.dat" is name of file to open, and  ios::out is optional
        If ( ! ob )
        {       cout<< "file not found or not opened";
                return;
        }
        -------
}
```

with this file object ob, we can do only write operations because it belongs to ofstream class.

Opening a file in both read and write mode

```
int  main()
{       fstream  ob;                    // file object 'ob' for reading + writing
        ob.open("stock.dat" ,  ios::in | ios::out );
        If ( ! ob)
        {       cout<< "file not found or not opened";
                return;
        }
        -------
}
```

Here file object ob is of type fstream class and file is opened with flags ios::in|ios::out, so we can do both read + write operations at a time on file.

How to read or write file data?

In C language, we have used fscanf() , fprintf() for text files, whereas fread(), fwrite() for binary operations. Similarly, in C++ we use i/o operators >>,<< for text file operations, this usage is same as used with cin,cout. The C++ functions read() and write() is used for binary file organization. We will learn through examples how to use in the programming.

closing a file

The function close() is used to close an opened file, after completion of updating file content, it must be closed. This operation involves in updating file contents on disk (saving back updated data to disk) and also releases/deletes buffer & file-stream in the RAM. If file is opened in read-mode, it does not required to store back in the disk, it only clears from the RAM.

Example1

Scans input values one by one from keyboard until last input is zero, writes each scanned value to disk. Later reads all such values one by one from disk and prints on the screen.z

Input from keyboard
20 30 40 45 56 23 34 0 ↲

output file ”sample.txt ”
20 30 40 45 56 23 34

```
#include<fstream>
#include<iostream>
using namespace std;
int main()
{       ofstream fout;     ifstream fin;   int n;
        fout.open("sample.txt");   // opening file in write-mode
        if( !fout )
        {       cout<<"error, file not opened";
                return 0;
        }
        while(1)
        {       cout<<"enter a value [at end 0]:";
                cin>>n;                 // reading int value from keyboard
                if(n==0)break;
                fout<<n<<" ";           // writs n value to disk using file object ‘fout’. (also adds space between each value )
        }
        fout.close();
        fin.open("sample.txt");      // opening same file in read-mode
        if(!fin)
        {       cout<<"error, file not opened";
                return 0;
        }
        while(1)
        {       fin>>n;                 // reading an integer value from file
                if(fin.eof()) break;    // if end-of-file reached then stops the program
                cout<<n<<"\n";          // showing n value on the screen
        }
        fin.close();
}
```

Example2

Above program in binary mode

```
#include<fstream>
#include<iostream>
using namespace std;
int main()
{       int n;
        ofstream  fout("sample.dat", ios::binary);     // opening file with constructor
        if( !fout )
        {       cout<<"error, file not opened";
                return 0;
        }
```

```
        while(1)
        {       cout<<"enter a value [at end 0]:";
                cin>>n;
                if(n==0)break;
                fout.write( (char*) &n, sizeof(n) );
        }
        fout.close();

        ifstream  fin( "sample.dat" , ios::binary );
        if(!fin)
        {       cout<<"error, file not opened";
                return 0;
        }
        while(1)
        {       fin.read( (char*)&n, sizeof(n) );
                if(fin.eof()) break;
                cout<<n<<"\n";
        }
    fin.close();
}
```

Example3

Copying odd numbers from one text file to another text file

Let our text file **"input.txt"** contained some even & odd numbers, now read numbers one by one from file and copy only **odd** numbers into another file called **"output.txt"**.

file name: "input.txt"
12 17 20 13 29 30 32 35 40 43 27 20 21 29 31 39 11 12 10 8 2 3

→

File name:"output.txt"
17 13 29 35 43 27 21 29 31 39 11 3

```
#include<fstream>
#include<iostream>
using namespace std;
 int  main()
{       ofstream fout;     ifstream fin;
        char sName[30], dName[30];  int n;
        cout<<"enter source & destination file names :";
        gets(sName);    gets(dName);
        fin.open(sName);
        fout.open( dName) ;
        if( !fin || !fout)
        {       cout<<"error, file(s) not found";
                return 0;
        }
        while(1)
        {       fin>>n;                         // reading an integer from file
                if( fin.eof()==true ) break;
                if(n%2==1) fout<<n<<" ";   // if odd number then writes to file
        }
        fin.close();   fout.close();            // saves onto disk
}
```

Example4

Counting upper/lower case alphabets, digits and others in a file

This program counts number of upper case alphabets, lower case alphabets, digits, words and lines in a given text file. It reads char by char from a given file and checks the each character and counts.

```
#include<fstream>
#sinclude<iostream>
using namespace std;
 int  main()
{       ofstream fout;   ifstream fin;
        int upperCount, lowerCount, digitCount, lineCount, wordCount;
        char sName[30], dName[30], ch;  int n;
        cout<<“enter source file name :”;
        gets(sName);
        fin.open(sName);
        if( !fin )
        {       cout<<"error, file not found";
                return 0;
        }
        upperCount=lowerCount=digitCount=lineCount=wordCount=0;
        while(1)
        {       fin>>ch;
                if( fin.eof() )  break;
                if( isupper(ch) ) upperCount++;
                else if( islower(ch) ) lowerCount++;
                else if( isdigit(ch) ) digitCount++;
                else if(ch==' ') wordCount++;
                else if(ch=='\n')
                {       wordCount++;
                        lineCount++;
                }
        }
        fin.close();
        cout<<"lower count =  "<<lowerCount;
        cout<<"upper count =  "<<upperCount;
        cout<<"digits count = “ << digitsCount;
        cout<<"words count = “ <<wordCount;
        cout<<”lines count = “<<lineCount;
}
```

Example5

Copying one file content to another file (file can be binary or text)

Note: the binary file organization works for both text/binary format files, this program reads byte by byte from source file and writes to target file, to read and store each byte value, the suitable data type is char.

```
#include<fstream>
#include<iostream>
using namespace std;
 int  main()
{       ofstream fout;   ifstream fin;
        char sName[30] , dName[30] , ch;
        cout<<“enter source & destination file names :”;
        gets(sName);    gets(dName);
```

```
        fin.open(sName , ios::binary);
        fout.open( dName , ios::binary) ;
        if( !fin || !fout )
        {       cout<<"error, file(s) not found";
                return 0;
        }
        while(1)
        {       fin.read( &ch , sizeof(ch) );          // reading char by char from file
                if( fin.eof() ) break;
                fout.write( &ch, sizeof(ch) );         // writing char by char to file
        }
        fin.close();  fout.close();
   }
```

Example6

Handling student details (in binary file format)

This program accepts student marks from keyboard and inserts each record into a file called “marks.dat”, later read back and process the result and shows on the screen.

Note: this is demo program, for binary file IO system; here class type is used.

input:
```
enter idno (0 to stop): 101↲
enter name:  Srihari↲
enter marks1 , marks2:  66  77↲
enter idno (0 to stop): 102↲
enter name:  Narahari↲
enter marks1 , marks2:  88  99↲
enter idno (0 to stop): 0↲
```

ouput:
```
idno    name        mark1   mark2   result
-----------------------------------------------------------------
100     Srihari      70      80      pass
101     Narahari     80      20      fail
```

```
#include<fstream>
#include<iostream>
using namespace std;
class Student
{       int  idno, marks1, marks2;
        char name[30];
        public:
        void scanMarksAndStoreToFile()
        {       ofstream file;
                file.open("student.dat", ios::binary|ios::app);       // ios::app → appends records to file
                if( !file )
                {       cout<<"file not opened";
                        return;
                }
                while(1)
                {       cout<<"\n enter idno (0 to stop):";
                        cin>>idno;
                        if(idno==0) break;        // 0 is end of input
                        cout<<"enter name:";
```

```
                fflush(stdin);
                cin>>name;
                cout<<"enter marks1 , marks2:";
                cin>>marks1>>marks2;
                file.write( (char*)this, sizeof(*this));
            }
        }
        void showFile()
        {       ifstream file;
                file.open("student.dat", ios::binary);
                cout<<"\nidno  name    marks1   marks2";
                cout<<"\n-------------------------------";
                if( !file )
                {       cout<<"file not opened";
                        return;
                }
                while(1)
                {       file.read((char*)this, sizeof(*this));
                        if(file.eof()) break;
                        cout<<"\n"<<idno<<"  "<<name<<"  "<<marks1<<" "<<marks2;
                }
                file.close();
        }
    };
    int main()
    {       Student ob;
            ob.scanMarksAndStoreToFile();
            ob.showFile();
    }
```

Example7

A menu driven program to handle employee records

This program manages employee information: it supports adding newly joined employee details, deleting relieving employee record, modifying salary, printing particulars.
Employee details are stored in a separate file called "emp.dat".
in this menu run program, the user can select his choice of operation.

Menu Run

```
--------------------------------
1. Add new employee
2. Delete record
3. Modify record
4. Print all records
0. Exit
        Enter choice [1,2,3,4,0]:
#include<stdio.h>
#include<iostream>
#include<fstream>
using namespace std;
class Employee              // employee class
{       int  empNo;
        char name[30];
        float salary;
        public:
```

```
        void appendRecord();
        void modifyRecord();
        void deleteRecord();
        void printRecord();   // fn proto-types
    };
    //  this append() function appends a record at end of file
    void  Employee::appendRecord()
    {       ofstream file;
        file.open("emp.dat", ios::binary|ios::app);
        if( !file )
        {       cout<<"\n Unable to open emp.dat file";
                return;
        }
        cout<<"\n enter employee no:";
        cin>>empNo;
        cout<<" Enter employee name:";
        fflush(stdin);
        cin>>name;
        cout<<" enter salary :";
        cin>>salary;
        file.write((char*)this , sizeof(*this));
        file.close();
        cout<<"\n successfully record added";
    }
// ---------------------------------------------------------------------------------------------------------------------
//  delete () function. Direct deletion of record is not possible from a file, alternative is, copies all records
// into another temp file except deleting record; later temp file is renamed with the original file name.
    void Employee::deleteRecord()
    {       ifstream fin;    ofstream fout;    Employee e;
        int eno, found=0, k ;
        fin.open("emp.dat", ios::binary );
        fout.open("temp.dat", ios::binary );
        if( !fin || !fout )
        {       cout<<"\n unable to open file";
                fin.close(); fout.close(); return;
        }
        cout<<"\n enter employee number to delete record :";
        cin>>eno;
        while(1)
        {       fin.read((char*)&e,sizeof(e));
                if(fin.eof()) break;
                if(eno==e.empNo)
                    found=1;     // record is found
                else
                    fout.write((char*)&e,sizeof(e));
        }
        if(found==1) cout<<"\n Record deleted success fully";
        else cout<<"\n Record Not found";
        fin.close(); fout.close();
        remove("emp.dat");                      // deletes old-file from disk
        rename("temp.dat","emp.dat");
    }
// ---------------------------------------------------------------------------------------------------------------------
```

```
//  Modifying data in a record. First, it searches for modifying record in a file, if record is not found then
// displays error message & returns. If found, then old-salary overwrites by new-salary of a record
        void Employee::modifyRecord()
        {       Employee e;                     fstream file;
                int found=0 , eno, k, pos;
                file.open("emp.dat",ios::binary|ios::in|ios::out);
                if(!file)
                {       cout<<"\n file not found ";
                        return;
                }
                cout<<"\n enter employee number:";
                cin>>eno;
                while(1)
                {       file.read( (char*)&e, sizeof(e));
                        if( file.eof() ) break;

                        if(eno==e.empNo)
                        {       found=1;        // record is found
                                break;
                        }
                }
                if(found==0) {  cout<<"\n Record not found"; return; }
                pos=file.tellg();               // this function returns the current position of read/write pointer
                pos=pos-sizeof(e);
                file.seekp(pos ,ios::beg);    /* move the file pointer one record position back,  after reading our search record ,
                the file pointer moves to next-record, so to replace our new record , we have to move the file pointer back.
                The above code moves like that. */
                cout<<"old salary :"<<e.salary;
                cout<<"enter new salary:";
                cin>>e.salary;
                file.write((char*)&e,sizeof(e));        // overwriting old record
                file.close();
               cout<<"\n address suceessfully modified";
        }
// ---------------------------------------------------------------------------------------------------------------------------------
// print function, prints all records
        void Employee::printRecord()
        {       Employee e;  int k, count=0;   ifstream file;
                file.open("emp.dat",ios::binary);
                if( !file )
                {       cout<<"\n file not found ";     return;
                }
                while(1)
                {       file.read( (char*)&e , sizeof(e) );
                        if( file.eof() ) break;
                        cout<<"\n employee number :"<<e.empNo;
                        cout<<"\n employee name :"<<e.name;
                        cout<<"\n Salary ::"<<e.salary;
                        count++;
                        cout<<"\n-----------------------------------------";
                }
                cout<<"\n"<< count<<" records found";
                file.close();
           }
```

```
int main()
{       int choice;
        Employee ob;
        while(1)        // loop to display menu continuously
        {       cout<<"\n\n=========================================================";
                cout<<"\n 1.append \n 2.delete \n 3.modify\n 4 print all\n 0.exit";
                printf("\n   Enter choice [1,2,3,4,0]:");
                cin>>choice;
                switch(choice)
                {       case 1:  ob.appendRecord(); break;
                        case 2:  ob.deleteRecord(); break;
                        case 3:  ob.modifyRecord(); break;
                        case 4:  ob.printRecord(); break;
                        case 0:  return 0;
                }
            }
            return 0;
      }
// ------------------------------------------------------------------------------------------------------------------------------------
```

Runtime Type Identification (RTTI)

Run-time type information (RTTI) is a mechanism that allows the type of an object to be determined during program execution. RTTI was added to the C++ language because many vendors of class libraries were implementing this functionality themselves. This caused incompatibilities between libraries. Thus, it became obvious that support for run-time type information was needed at the language level.

C++ provides RTTI through a set of operators and functions. There are three main C++ language elements to run-time type information:

The 'typeid' operator: used for identifying the exact type of an object. This is an operator introduced in C++98 standard version, associated with 'type_info' class.
The 'type_info' class: Used to hold the type information returned by the typeid operator.
The 'dynamic_cast' operator: used to cast one type to another type.
The header file <typeinfo> should be included to use typeid operator and typeinfo class. eg: #include<typeinfo>
This typeid operator and typeinfo class are defined in namespace 'std'.

typeid(): returns an object of 'typeinfo' class. It contains information about data-type of an object.
The member function name() gives data type name of a given object. For example

typeid(int).name() → i
typeid(float).name() → f
typeid(char).name() → c

```
#include<typeinfo>
#include<iostream>
using namespace std;
class TestClass { };
int  main()
{       TestClass  ob;
        cout <<"\n output is:"<<typeid(int).name();   → i
        cout <<"\n output is:"<<typeid(float).name();   → f
        cout <<"\n output is:"<<typeid(char).name();   → c
        cout <<"\n output is:"<<typeid(TestClass).name();   → TestClass
        cout <<"\n output is:"<<typeid(ob).name();   → TestClass
}
```

```
#include<typeinfo>
#include<iostream>
class  A { };
class  B { };
int  main()
{       A   ob1;   B  ob2;
        if ( typeid(ob1) == typeid(ob2) )   std::cout <<"\n ob1 and ob2 are of the same type";
        else    std::cout << "\n ob1 and ob1 are not  same types";
}
```

output: ob1 and ob1 are not same types

---

type_info class

The typeid() operator returns an object of the 'type_info' class. This class facilitates RTTI and provides the following functions

name() : returns a string representing the name of the type.

operator== : compares two type_info objects for equality.

operator!= : compares two type_info objects for inequality.

before() : tells the order of classes before or after found in the program, returns bool.

```
#include <iostream>
#include <typeinfo>
class Base {  };
class Derived : public Base {};
using namespace std;
int main()
{   Base b;   Derived d;
    Base* ptr = &d;
    cout << "Type of b: " << typeid(b).name() << "\n";
    cout << "Type of *ptr: " << typeid(*ptr).name() << "\n";
    cout << "Type of ptr: " << typeid(ptr).name() << "\n";
    cout<< typeid(b).before( typeid(d) );
    return 0;
}
```

Output:

```
Type of b: Base
Type of *ptr: Derived
Type of ptr: Base*
1 (true)  → b-class declared after d-class
```

We have 3 casting operators related to RTTI. dynamic_cast, static_cast, const_cast, reinterpret_cast

1. **dynamic_cast**
   - Used for safely downcasting pointers or references in polymorphic class hierarchies.
   - It performs runtime checks to ensure the cast is valid. If the cast fails (for pointers), it returns nullptr. For references, it throws a std::bad_cast exception.
2. **static_cast**
   - Not directly related to RTTI but can be used in scenarios where the types involved are known at compile-time.
   - It performs compile-time checks and doesn't involve runtime type information. It's not safe for downcasting polymorphic types unless the cast is guaranteed to be correct.
3. **const_cast**
   - Used to add or remove const qualifier from a pointer or reference.
   - It does not rely on RTTI but can be used in conjunction with RTTI in some cases when you need to remove const from an object in polymorphic types.
4. **reinterpret_cast**
   - Also not directly tied to RTTI but allows casting between unrelated types, such as between a pointer to a type and a pointer to another type, irrespective of the actual type of the object.
   - It doesn't check for type compatibility and can lead to undefined behavior if misused.

dynamic_cast: used for safely downcasting pointers or references in dynamic polymorphism concept. It used to get address of child class object from a parent class pointer. That is, it returns or converts parent class type to child class type. (down casting)

Syntax: dynamic_cast<T*>(expression) // for pointers
dynamic_cast<T&>(expression) // for references

```
#include <iostream>
#include <typeinfo>
class Base
{       public:
        virtual void  show()
        {       std::cout<<"Base";
        }
};
class Derived : public Base
{       public:
        void show()
        {       std::cout<<"Derived";
        }
};
int main()
{       Base* basePtr = new Derived();
        Derived* derivedPtr = dynamic_cast<Derived*>(basePtr);  // observe, down casting
        if(derivedPtr)
                derivedPtr->show();      // Safe to call Derived-specific functions
        else
                std::cout << "dynamic_cast failed\n";
        basePtr->show();
        delete basePtr;
        return 0;
}
```

output: Derived
Derived

---

static_cast: this is compile-time cast same as c-style casting. The difference is, compiler reports error if casting is in valid. It does things like implicit conversions between types (such as int to float, or pointer to void*), and it can also call explicit conversion functions.

syntax: static_cast <conversion_type> (value)

```
#include <iostream>
using namespace std;
int main()
{       float x = 3.5;   int  y;
        y = x;                                  // c-style implicit conversion.
        cout << "the value of y: " << y;

        y = static_cast<int>(x);                // using static_cast for float to int
        cout << endl<<"the value of y: " << y;
}
```

---

const_cast: used to add or remove constant or volatile modifier of a variable.

syntax: const_cast<type name>(expression)

```
int  main()
{       const int x = 50;
        const int   *y = &x;
        cout<<endl<<old value is"<<*y;
        *y=100;                                 // here *y is constant, can't be modified.
        int   *z = const_cast<int*> (y);
        *z=100;
        cout<< endl<<"new value is"<<*y;
}
```

Output: old value is 50
new value is 100

reinterpret_cast: It is used to access low level binary data between two objects (streams) using pointers. It converts a pointer of one data type to another type.

```
data_type  *target_pointer  =  reinterpret_cast<data_type*> (source_pointer);
```

```
#include <iostream>
int main()
{       int  *pi = new int(65);
        char  *pc = reinterpret_cast<char*>(pi);
        cout << *pc << endl;
        pc++;
        cout << *pc << endl;
}
```

Runtime Type Identification (RTTI)

Run-time type information (RTTI) is a mechanism that allows the type of an object to be determined during program execution. RTTI was added to the C++ language because many vendors of class libraries were implementing this functionality themselves. This caused incompatibilities between libraries. Thus, it became obvious that support for run-time type information was needed at the language level.

C++ provides RTTI through a set of operators and functions. There are three main C++ language elements to run-time type information:

The 'typeid' operator: used for identifying the exact type of an object. This is an operator introduced in C++98 standard version, associated with 'type_info' class.
The 'type_info' class: Used to hold the type information returned by the typeid operator.
The 'dynamic_cast' operator: used to cast one type to another type.
The header file <typeinfo> should be included to use typeid operator and typeinfo class. eg: #include<typeinfo>
This typeid operator and typeinfo class are defined in namespace 'std'.

typeid(): returns an object of 'typeinfo' class. It contains information about data-type of an object.
The member function name() gives data type name of a given object. For example

typeid(int).name() → i
typeid(float).name() → f
typeid(char).name() → c

```
#include<typeinfo>
#include<iostream>
using namespace std;
class TestClass { };
int  main()
{       TestClass  ob;
        cout <<"\n output is:"<<typeid(int).name();   → i
        cout <<"\n output is:"<<typeid(float).name();   → f
        cout <<"\n output is:"<<typeid(char).name();   → c
        cout <<"\n output is:"<<typeid(TestClass).name();   → TestClass
        cout <<"\n output is:"<<typeid(ob).name();   → TestClass
}
```

```
#include<typeinfo>
#include<iostream>
class  A { };
class  B { };
int  main()
{       A   ob1;   B  ob2;
        if ( typeid(ob1) == typeid(ob2) )   std::cout <<"\n ob1 and ob2 are of the same type";
        else    std::cout << "\n ob1 and ob1 are not  same types";
}
```

output: ob1 and ob1 are not same types

---

type_info class

The typeid() operator returns an object of the 'type_info' class. This class facilitates RTTI and provides the following functions

name() : returns a string representing the name of the type.

operator== : compares two type_info objects for equality.

operator!= : compares two type_info objects for inequality.

before() : tells the order of classes before or after found in the program, returns bool.

```
#include <iostream>
#include <typeinfo>
class Base {  };
class Derived : public Base {};
using namespace std;
int main()
{   Base b;   Derived d;
    Base* ptr = &d;
    cout << "Type of b: " << typeid(b).name() << "\n";
    cout << "Type of *ptr: " << typeid(*ptr).name() << "\n";
    cout << "Type of ptr: " << typeid(ptr).name() << "\n";
    cout<< typeid(b).before( typeid(d) );
    return 0;
}
```

Output:

```
Type of b: Base
Type of *ptr: Derived
Type of ptr: Base*
1 (true)  → b-class declared after d-class
```

We have 3 casting operators related to RTTI. dynamic_cast, static_cast, const_cast, reinterpret_cast

1. **dynamic_cast**
   - Used for safely downcasting pointers or references in polymorphic class hierarchies.
   - It performs runtime checks to ensure the cast is valid. If the cast fails (for pointers), it returns nullptr. For references, it throws a std::bad_cast exception.
2. **static_cast**
   - Not directly related to RTTI but can be used in scenarios where the types involved are known at compile-time.
   - It performs compile-time checks and doesn't involve runtime type information. It's not safe for downcasting polymorphic types unless the cast is guaranteed to be correct.
3. **const_cast**
   - Used to add or remove const qualifier from a pointer or reference.
   - It does not rely on RTTI but can be used in conjunction with RTTI in some cases when you need to remove const from an object in polymorphic types.
4. **reinterpret_cast**
   - Also not directly tied to RTTI but allows casting between unrelated types, such as between a pointer to a type and a pointer to another type, irrespective of the actual type of the object.
   - It doesn't check for type compatibility and can lead to undefined behavior if misused.

dynamic_cast: used for safely downcasting pointers or references in dynamic polymorphism concept. It used to get address of child class object from a parent class pointer. That is, it returns or converts parent class type to child class type. (down casting)

Syntax: dynamic_cast<T*>(expression) // for pointers
dynamic_cast<T&>(expression) // for references

```
#include <iostream>
#include <typeinfo>
class Base
{       public:
        virtual void  show()
        {       std::cout<<"Base";
        }
};
class Derived : public Base
{       public:
        void show()
        {       std::cout<<"Derived";
        }
};
int main()
{       Base* basePtr = new Derived();
        Derived* derivedPtr = dynamic_cast<Derived*>(basePtr);  // observe, down casting
        if(derivedPtr)
                derivedPtr->show();      // Safe to call Derived-specific functions
        else
                std::cout << "dynamic_cast failed\n";
        basePtr->show();
        delete basePtr;
        return 0;
}
```

output: Derived
Derived

---

static_cast: this is compile-time cast same as c-style casting. The difference is, compiler reports error if casting is in valid. It does things like implicit conversions between types (such as int to float, or pointer to void*), and it can also call explicit conversion functions.

syntax: static_cast <conversion_type> (value)

```
#include <iostream>
using namespace std;
int main()
{       float x = 3.5;   int  y;
        y = x;                                  // c-style implicit conversion.
        cout << "the value of y: " << y;

        y = static_cast<int>(x);                // using static_cast for float to int
        cout << endl<<"the value of y: " << y;
}
```

---

const_cast: used to add or remove constant or volatile modifier of a variable.

syntax: const_cast<type name>(expression)

```
int  main()
{       const int x = 50;
        const int   *y = &x;
        cout<<endl<<old value is"<<*y;
        *y=100;                                 // here *y is constant, can't be modified.
        int   *z = const_cast<int*> (y);
        *z=100;
        cout<< endl<<"new value is"<<*y;
}
```

Output: old value is 50
new value is 100

reinterpret_cast: It is used to access low level binary data between two objects (streams) using pointers. It converts a pointer of one data type to another type.

```
data_type  *target_pointer  =  reinterpret_cast<data_type*> (source_pointer);
```

```
#include <iostream>
int main()
{       int  *pi = new int(65);
        char  *pc = reinterpret_cast<char*>(pi);
        cout << *pc << endl;
        pc++;
        cout << *pc << endl;
}
```

Exercise programs

1) Complete the code of following Circle class according to function calls given in the main() function.

```
class  Circle
{   ------------
    ------------
};
int  main()
{       Circle ob(5);       // 5 is the radius value of circle, set through constructor.
        ob.find();          // find  area and circumference of circle at this function.
        ob.show();          //  print area and circumference at this function.
}
```

---

2) Implement a class called **Employee ,** where write functions setSalary(), calculateTax(), printTax().

tax calculation process:

if salary<=10000 then tax=0

if salary>10000 and salary<=20000 then tax is 5% on salary

if salary>20000 then tax is 8% on salary.

```
class  Employee
{   ------------
    ------------
};
int  main()
{       Employee  ob;
        ob.setSalary ( 7000 );      // this set() function sets the 7000/- to salary
        ob.calculateTax();          // calculate tax as above said
        ob.printTax();              // print tax which is calculated above function.
}
```

---

3) Implement a class called ***MyDate*** with 2 functions scan() and isEqual(), the scan() function reads values from keyboard, whereas isEqual() compares given two dates equal or not? returns bool value, the main() function is as follows

```
class  MyDate
{   ------------
    ------------
};
int  main()
{       MyDate ob1, ob2;
        ob1.scan();
        ob2.scan();
        bool flag = ob1.isEqual(ob2);
        if(flag==1)  cout<<"equal";
        else cout<<"not equal";
}
```

---

4) Implement a class called ***MyDate*** with 2 functions set() and isEqual(). The set() function sets the input values to objects which are scanned from keyboard at man() function, whereas isEqual() compares equality of two dates and returns bool value, the main() function is as follows

```
class  MyDate
{   ------------
    ------------
};
int main()
{       MyDate ob1, ob2;
        int  d,m,y;
        cout<<"enter date1 :";
        cin>>d>>m>>y;
        ob1.set(d,m,y);
        cout<<"enter date2 :";
        cin>>d>>m>>y;
        ob2.set(d,m,y);
        bool flag=ob1.isEqual(ob2);
        if(flag==true)  cout<<"equal";
        else cout<<"not equal";
}
```

---

5) Implement a class called ***MyDate*** where compare two dates. Initialize the date objects with sample values through constructor and finally print equal or not?

```
class  MyDate
{   ------------
    ------------
};
int main()
{       MyDate ob1(12,3,2024) , ob2(10,12, 2025);
        bool  flag = ob1.isEqual(ob2);
        if(flag==true)  cout<<"equal";
        else cout<<"not equal";
}
```

---

6) Implement a class called **Player**, here it holds the player name and score as data members and it counts the score and also displays when it is required. Based on given main() fn, implement a class and its functionality. The main is as follows

```
class  Player
{       -----------------
        -----------------
};
int  main()
{       Player ob1("Ram");   // the object 'ob1' for player-1, initialize  player name with  "Ram"  and score with 0.
        Player ob2("Ravi");   // the object 'ob2' for player-2, initialize  player name with  "Ravi"  and score with 0.
        ob1.addScore(10);
        ob2.addScore(20);
        ob1.addScore(7);
```

```
        ob2.addScore(12);
        ob1.showScore();     //  Ram score is 17
        ob2.showScore();     //  Ravi score is 32
        int total= ob1.getScore()+ob2.getScore();     //  getScore() returns score of player
        printf("total score is %d ", total );         // displaying  total score two players.
    }
```

---

7) Implement a class called **My*Time,*** where take two sample times like employee working time in two shifts, now find total time worked in 2 shifts, the main() fn is as follows

```
    class MyTime
    {       private:  int h,m,s;
            public:   Time() { … }
                      Time( int h, int m, int s ) { ... }
                      MyTime  add( MyTime ob2 ) { … }
                      void show() { … }
    };
    int  main()
    {       MyTime ob1(5,30,50) , ob2(4,50,50),  ob3;
            ob3 = ob1.add(ob2) { … }
            ob3.show() { … }  // output:  10:21:40
    }
```

---

8) Implement a class called MyTime, which includes two functions: add() and increment().
Both functions increment the given time by N seconds, but there is a slight difference between them:
The add() function adds N seconds to a temporary object (not modifying the original object) and returns the result. The increment() function adds N seconds directly to the given object, modifying it.
The following main() function provides a clearer idea:

```
class MyTime
{       private:  int h,m,s;
        public:   void  set(int h, int m, int s) {…}
                  MyTime  add(int n) {…}
                  void  increment(int n) {…}
                  void show() {…}
};
int main()
{       MyTime ob1, ob2;
        ob1.set( 5,40,50 );
        ob2 = ob1.add( 3600 );    //  3600 seconds (1 hour)
        ob2.show();               //  6:40:50
        ob1.show();               //  5:40:50  → notice this, no change of values in this object
        ob1.increment(3600);      // adds 3600 seconds to 'ob1' itself.
        ob1.show();               // 6:40:50  → notice the change of values in this object.
}
```

---

9) Implement a class called **PrintNumbers** where write functions setN(), print1toN(), print1toNOdds(), printTable() and printPrimes(). The ADT class is as given below,

```
class  PrintNumbers
{       private:  int N;
        public:   void setN(int N) { … }
                  void print1toN() { … }          // prints 1,2,3,4, ….N
                  void print1toNOdds() { … }      // prints 1,3,5,7,  … N
                  void printTable() { … }         // if N=8 then  8*1=8, 8*2=16, 8*3=24, … upto 10 times
                  void printPrimes() { … }        // print all primes between 1 to N
};
int  main()
{       PrintNumbers ob;
        ob.setN(8);
        ob.print1toN();
        ob.print1toNOdds();
        ob.printTable();
        ob.printPrimes();
}
```

---

10) Implement a class called “Complex” where take two sample complex numbers and print addition of them, the Abstract class as given below

```
class Complex
{       int real, img;
        void set(int real, int img) { … }
        void show() { … }
        Complex  add(Complex ob2)
        {   Complex t;
            t.real=real+ob2.real;
            t.img=img+ob2.img;
            return t;
        }
};
void main()
{   Complex ob1,ob2,ob3;
    ob1.set(4,5);
    ob2.set(7,8);
    ob3=ob1.add(ob2);
    ob3.show();                 //  11 + 13i
}
```

---

11) Implement a class called 'Student', where write functions called scan() and calculate();

calculate the result as given below:

total=mark1+mark2;
average=total/2;
if student got <35 in mark1 or mark2 then result="failed"
or else, result="A-grade/B-grade/C-grade" based on average
if average>=60 result="A-grade"
if average>=50 and <60 result="B-grade"
if average<50 result="C-grade"

```
class  Student
{       private:  int idno, mark1, mark2, total;
                  float avg;
                  char *result, name[30]
        public:
            void scan() { … }
            void calculate() { … }
            void printResult()
            {    printf("output: %d  %s %d %d %d %f %s",idno,name,marks1,makrs2,total,average,result);
            }
};
int main()
{       Student ob;
        ob.scan();
        ob.calculate();
        ob.printResult();
}
```

---

12) Extend above program to handle 5 students details (idno, name, marks1, marks2, total, average & result) and show result of 5 students as shown in main() fn.

```
int main()
{   Student ob[5];      // let us have 5 students, array of 5 objects
    ob[0].set(101 , "Hari",  82, 67);          // set( idno, name, marks1, marks2);
    ob[1].set(102 , "Seetha", 94, 77);
    ob[2].set(103 , "Ramya", 70, 80);
    ob[3].set(104 , "Laxmi", 54, 79);
    ob[4].set(105 , "Balu", 56, 56);
    printf("\n indo, name, mark1, mark2, total, average, result");
    printf("\n==========================================");
    for(i=0; i<5; i++)
        ob[i].showResult();
}
```

---

13) Implement a class called **Bank** with following functionalities

```
class Bank
{       float balance;
        Bank() { balance=0; }           // this constructor function automatically called when object is created
        void showBalance() { … }        // show balance on the screen
        void deposit( float amount ) { … }      // add amount to balance & show message "successfully deposited"
        void withdraw( float amount ) { … } // if amount>balance then print error message called "transaction
                failed/insufficient funds" otherwise subtract amount from balance & print "successfully withdrawn"
};
int  main()
{       Bank ob;
        ob.showBalance();       // current balance = 0/-
        ob.deposit(1200);       // successfully deposited = 1200/-
        ob.showBalance();       // current balance = 1200/-
        ob.withdraw(700);       // successfully withdrawn = 700/-
        ob.showBalance();       // current balance = 500/-
        ob.withdraw(3200);      // transaction failed, insufficient funds, balance is 500/- only
}
```

Also test above class using following main() fn (this is menu-run program)

```
int main()
{   Bank ob;   int ch;
    while(1)
    {   printf("\n 1. Deposit  \n2.Withdraw  \n3.Show balance \n0.exit");
        printf("\n Enter choice [1|2|3|4|0] :");
        scanf("%d", &ch);
        switch(ch)
        {   case 1:   printf("enter deposit amount :");
                      scanf("%f", &amount);
                      ob.deposit(amount);
                      break;
            case 2:  printf("enter withdrawal amount:");
                      scanf("%f", &amount);
                      ob.withdraw(amount);
                      break;
            case 3:   ob.showBalance();    break;
            case 0:   exit(0);
        }
    }
}
```

---

14) Implement a class called Game where two players play this game, here add score as given below

```
class Game
{  ----------
   ----------
};
int  main()
{  Game ob("Ram", "Ravi");      // this single object holds two players name & their score.
   ob.addScore("Ram", 5);       // first argument is player name and second argument is score he made.
   ob.addScore("Ravi", 2);      //  Ravi made 2 points score
   ob.addScore("Ravi", 4);      //  Ravi made 4 points score
   ob.addScore("Ram", 1);       // first argument is player name and second argument is score he made.
   ob.addScore("Ram", 7);       //  Ram  made 7 points score
   ob.addScore("Rani", 3);      //  error, invalid player name Rani
   ob.showScore("Ram");         // Ram's score is: 13
   ob.showScore("Ravi");        //  Ravi's score is: 6
   ob.showScore();              // displays two players score:  Ram score is 13, Ravi score is 5
}
```

---

15) Implement a class called **My*Time,*** where increment 'N' seconds to given time and also maintain AM/PM.

```
class Time
{       ---------
        ---------
};
int main()
{  Time ob;
   ob.setTime(11,40,50,"AM");
   ob.incrementTime(7200);
   ob.showTime();              //output is 1:40:50 PM
}
```

---

16) Implement a class called **StudentDatabase** where maintain list of 10 students details, the data like student-name and phone-number. Now our functionality is to search and return phone-number or student-name based given argument. For this time, let us consider student-name or phone-number is not repeated, that is, every student name is unique and phone number is also unique. Take student-name as string & phone as long int type. The main() fn is as follows

```
int  main()
{  StudentDataBase ob;
   ob.addStudent("Ram", 9440030405);
   ob.addStudent("Ravi", 9440112113);
   ob.addStudent("Rahim", 8500117118);
   ob.addStudent("Ibrahim", 9494113114);
   ob.showDetails("Ram");  → 9440030405
   ob.showDetails(9440030405L);  → Ram
   ob.showDetails("Rahim");  → 8500117118
   ob.showDetails("xyz");  → student  not found
}
```

---

17) This is an extension to above program, but here consider student-name or phone-number can be duplicated. The main() fn is as follows

```
int main()
{   StudentDataBase ob;
    ob.addStudent("Ram", 9440030405);
    ob.addStudent("Ravi", 9440112113);
    ob.addStudent("Rahim", 8500117118);
    ob.addStudent("Laxmi", 9440030405);
    ob.addStudent("Ibrahim", 9440030405);
    ob.addStudent("Ram", 9440111222);
    ob.showDetails("Ravi");   → 9440112113
    ob.showDetails("9440030405");   → Ram , Laxmi
    ob.showDetails("Rahim");  → 8500117118
    ob.showDetails("Ram");  → 9440030405, 9440111222
}
```

---

18) Implement a class called ***"List",*** where write functions to insert a value, to delete a value, to search a value, and to print all values. The abstract class as given below

```
class List
{       int a[100], count;
    public:
        List() { count=0; }          // this is constructor, automatically called when object is created.
        void insert(int v) {…}       // inserts value in the array
        void delete(int  v) {…}      // deletes value in the array, if not found then display error message
        int search(int v) {…}        // to search for value in the array, if found then return 1, otherwise return 0.
        void print() { … }           // to print all values in the array.
};
int  main()
{   List ob;
    ob.insert(12);
    ob.insert(10);
    ob.insert(7);
    ob.insert(8);
    ob.print();       // 12, 10, 7, 8
    ob.delete(7);
    ob.print();       // 12, 10,  8
    if(ob.search(12)==1)   printf("element found");
    else    printf("element not found");
}
```

---

19) Implement classes “Student & School” where maintain students ‘fee’ database. Here every student has idno, name & fee, record list of all students details as given in main() fn.

```
class Student
{       int idno;
        char name[30];
        float fee;
        public:
        Student(int idno, char name[], float fee) { … }     // parameterized constructor
        int getIdno()
        {    return idno;
        }
        void show() { … }
};

class School
{       Student *st[10];
        int count;
    public:
        Student() { count=0;  }
        void addRecord(int idno, char name[], float fee)
        {   int i;
            for(i=0; i<count; i++)
            {   if( p[i]->getIdno()==idno )
                {   printf(“error, student %d already paid”, idno);
                    return;
                }
            }
            Student *a=new Student(idno, name, fee);
            p[count]=a;
            count++;
        }
        void show(int idno) { …}  // display particular student details like name, fee paid
        void showAll() { … }         // display all details of students
};

int main()
{   School ob;
    ob.addRecord(105,”Laxmi”, 300.00);        // 105 is idno, Laxmi is name, 300.00 is fee
    ob.addRecord(101,”Ram”, 150.00);
    ob.addRecord(100,”Ravi”, 250.00);
    ob.addRecord(105, “Laxmi”, 300.00);       // error, already paid
    ob.showAll();       //  displays all student records
    ob.show(101);      // op: { 101, “Ram”, 150.00 }
}
```

---

20) Implement a class “Bus” where reserve/un-reserve with 10 seats capacity; the classes as given below.

```
class Passenger
{       int seatNo, age;
        string name;
        public:
        Passenger(int seatNo, char name[], int age)  {… }  // parameterized constructor
        int getSeatNo()
        {   return seatNo;
        }
        void show() { … }
};

class Bus
{       Pasenger *p[10];
        int count;
        public:
        Bus() { count=0; }
        void reserve(int seatNo, char name[], int age)
        {   int i;
            for(i=0; i<count; i++)
                if( p[i]->getSeatNo()==seatNo)
                {   printf(“error, seat already reserved”);
                    return;
                }
                Passenger *a=new Passenger(seatNo, name, age);
                p[count]=a;
                count++;
        }
        void unReserve(int seatNo) { ... }
        void show(int seatNo) {… }  // display particular passenger details like name , age.
        void showAll() { … }              // display all details of passengers
};

int main()
{   Bus ob;
    ob.reserve(5,”Laxmi”, 55);
    ob.reserve(15,”Ram”, 15);
    ob.reserve(10,”Ravi”, 25);
    ob.reserve(5, “Rahim”, 44);    // error, seat no:5 already reserved
    ob.showAll();          // op: seats 5, 15, 25  are reserved and 7 seats left free
    ob.unReserve(15);   // seat 15 to un reserve
    ob.showAll();          //op: seat 5, 25  are reserved and 8 seats left free
    ob.show(5);            // op: {5,“Laxmi”,55}
}
```

---

21) Implement a class called Stack where maintain list of values (array) with following operations.
Stack is nothing but list of values where we do only two operations (push/pop).
1) write push() function, it inserts a value at the end of array called top of the stack.
2) write pop() function, it delete & returns top element of the stack (lastly inserted value)
3) both push() and pop() function handles error, stack full(overflow), stack empty(underflow)

```
class Stack
{   private:  int a[10],  count;
    public:
            Stack() { count=0;}   // initially no elements in stack
            void push(int v);
            int pop();
            void show();          // prints all elements in the stack.
            int emptyStack();     // returns bool value 1/0
};
int main()
{   Stack ob;
    ob.push(10);
    ob.push(22);
    ob.push(13);
    ob.push(9);
    ob.show();            //  10, 22, 13, 9
    k=ob.pop();
    printf("popped value is: %d", k); // popped values is: 9
    k=ob.pop();
    printf("popped value is %d", k);  // popped values is: 13
    k=ob.pop();
    printf("popped value is %d", k);  // popped values is: 22
}
```

---

22) Linked list data structure

```
#include<iostream>
#include<stdio.h>
class Node
{    public:
     int data;
     Node *next;
     Node() { data=0; next=NULL; }
     Node(int data) { this->data=data; next=NULL; }
     Node* getNext() { return next; }
     int getData(){ return data; }
     void insertNext(int data)
     {   Node *temp;
        temp=new Node(data);
        temp->next=this->next;
        this->next=temp;
     }
  };
```

```
class LinkedList
{    Node *head;
     public:
     LinkedList()
     {       head=new Node();        // creating header node, it is dummy in this program.
     }
     void insertFirst(int data)
     {       head->insertNext(data);
     }
    void insertLast(int data)
    {       Node *temp;
            temp=head;
            while(temp->next!=NULL)
                    temp=temp->next;
            temp->insertNext(data);
    }
    void showList()
    {       Node *temp;
               temp=head->next;
               printf("\n list is :");
            while(temp)
            {       printf("%d ", temp->data);
                    temp=temp->next;
            }
     }
     void deleteFirst()
     {      Node *temp;
            if(head->next!=NULL)
            {       temp=head->next;
                    head->next=head->next->next;
            }
            delete temp;
    }
    void deleteLast()
    {       Node *prev,*temp;
            temp=head;
            if(head->next==NULL)
                    return;
            while(temp->next!=NULL)
            {       prev=temp;
                    temp=temp->next;
            }
            prev->next=NULL;
            delete temp;
     }
};
```

```
int main()
{       LinkedList   ob;
        ob.insertFirst(3);
        ob.insertFirst(2);
        ob.insertFirst(1);
        ob.showList();
        ob.deleteLast();
        ob.showList();
        return 0;
}
```

---

23) Date and Time classes

First look at the main() function and then check the code in the classes.

```
#include<iostream>
using namespace std;
class Date
{  int day,month,year;
   public:

   void setDate( int d,int m, int y)
   {  day=d; month=m; year=y;
    }

    void showDate()
   {  cout<<"\n date is::"<<day<<"/"<<month<<"/"<<year;
   }

   void incrementDate(int n)
   {  int a[]={0,31,28,31,30,31,30,31,31,30,31,30,31};
     int i;
     if(year%4==0) a[2]=29;
     for(i=0; i<n; i++)
     {  day++;
       if(day>a[month])
       {  day=1; month++;
         if(month==13)
         {  year++;
           month=1;
           if(year%4==0)
              a[2]=29;
          }
       }
     }
    }
   };
```

```
class Time
{   protected:
    int hour,min,sec;
    char *amPm;
    public:
    void setTime(int h, int m, int s, char *amPm)
    {       hour=h; min=m; sec=s;
            if( *amPm=='a' || *amPm=='A')
                this->amPm="AM";
            else this->amPm="PM";
    }
    void showTime()
    {       cout<<"\ntime is::"<<hour<<" : "<<min<<" : "<<sec<<" : "<<amPm;
    }

    int incrementTime(int n)
    {       int prev,days=0;
            days=n/(24*3600);           // counting days if hr>24
            n=n%(24*3600);              // if more than one day
            prev=hour*3600+min*60+sec;
            if(prev<=12*3600 && prev+n>12*3600)
            {   if(*amPm=='P') amPm="AM";
                else amPm="PM";
                days++;
            }
            n=n+prev;   hour=n/3600;
            n=n%3600;  min=n/60; sec=n%60;
            if(hour>12) hour=hour%12;
            return days;
    }
};

class DateTime
{   Date dt;     Time tt;
    public:
    DateTime(int day, int month, int year, int h,int m,int s, char *amPm)
    {       dt.setDate(day,month,year);
            tt.setTime(h,m,s,amPm);
    }
    void incrementTime(int n)
    {       int k=tt.incrementTime(n);
            dt.incrementDate(k);
    }
    void incrementDate(int n)
    {       dt.incrementDate(n);
    }
```

```
    void showDateTime()
    {   dt.showDate();
        tt.showTime();
    }
};

    int main()
    {   DateTime ob(11,5,2024, 11,4,50,"PM");
        ob.showDateTime();
        ob.incrementTime( 7200 );      // 2hours
        cout<<"\n\n date and time is, after adding 2hr ";
        ob.showDateTime();
        ob.incrementDate(2);
        cout<<"\n\n date and time is, after adding 2days ";
        ob.showDateTime();
   }
```

The above code can be written using more than one level of inheritance. Complete the code yourself.

```
        Date Date
        {
        };
        Date Time :  public Date
        {
        };
        Date DateTime: public Time
        {
        };
        int  main()
        {
        }
```
