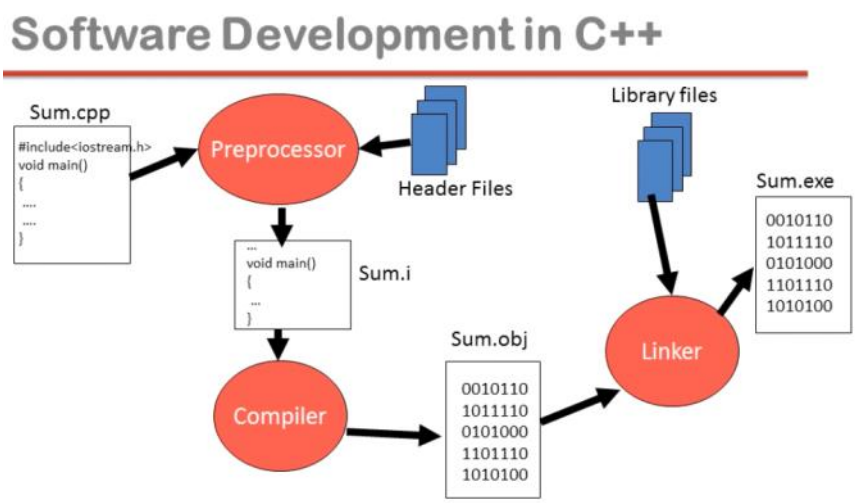
Basics

Differences between C, C++ and Java

Metrics	C	C++	Java
Programming Paradigm	Procedural language	Object-Oriented Programming (OOP)	Pure Object Oriented Oriented
Origin	Based on assembly language	Based on C language	Based on C and C++
Developer	Dennis Ritchie in 1972	Bjarne Stroustrup in 1979	James Gosling in 1991
Translator	Compiler only	Compiler only	Interpreted language (Compiler + interpreter)
Platform Dependency	Platform Dependent	Platform Dependent	Platform Independent
Code execution	Direct	Direct	Executed by JVM (Java Virtual Machine)
Approach	Top-down approach	Bottom-up approach	Bottom-up approach
File generation	.exe files	.exe files	.class files
Pre-processor directives	Support header files (#include, #define)	Supported (#header, #define)	Use Packages (import)
keywords	Support 32 keywords	Supports 63 keywords	50 defined keywords
Datatypes (union, structure)	Supported	Supported	Not supported
Inheritance	No inheritance	Supported	Supported except Multiple inheritance
Overloading	No overloading	Support Function overloading (Polymorphism)	Operator overloading is not supported
Pointers	Supported	Supported	Not supported
Allocation	Use malloc, calloc	Use new, delete	Garbage collector
Exception Handling	Not supported	Supported	Supported
Templates	Not supported	Supported	Not supported
Destructors	No constructor neither destructor	Supported	Not supported
Multithreading/ Interfaces	Not supported	Not supported	Supported
Database connectivity	Not supported	Not supported	Supported
Storage Classes	Supported (auto, extern)	Supported (auto, extern)	Not supported

Software Development In C++

The source code written by programmers is stored in the file *program.cpp* . This file is then processed by preprocessors and an expanded source code file is generated named *program.i* . This expanded file is compiled by the compiler and an object code file is generated named *program.obj* . Finally, the linker links this object code file to the object code of the library functions to generate the executable file *program.exe* . There is one more file *program.bak* which is backup file if the main program.cpp file gets corrupted.
{ .cpp & .bak } -> text files
{ .obj & .exe } -> binary files




```
----------------------------------------------------Diagram 1-----------------------------------------------
[Source Code] ---> Compiler ---> [Object code] --*
                                                 |
[Source Code] ---> Compiler ---> [Object code] --*--> Linker --> [Executable] ---> Loader
                                                 |                                    |
[Source Code] ---> Compiler ---> [Object code] --*                                    |
                                                 |                                    |
                                [Library file]---*                                    V
                                                                    [Running Executable in Memory]

------------------------------------------------------------------------------------------------------------
```

```
----------------------------------------------------Diagram 2-----------------------------------------------
                                                 Editor Or Ide  (write source code)
                                                      |
                                                      |   (.cpp, .h) Source Code, header files
                                                      V
                                                 Preprocessor
                                                      |
                                                      |   (*.i) Included Files, Replaced Symbols
                                                      V
                                                 Compiler
                                                      |
                                                      |   (*.s) assembly code
                                                      V
                                                 Assembler
                                                      |
                                                      |   (*.o) Object Code
                                                      V
Static Libraries (.lib, .a)------------>  Linker
                                                      |
                                                      |   (.exe)
                                                      V
Dynamic Libraries (.dll, .so)---------->  Loader
                                                      |
                                                      |
                                                      V
                                                 Operating System
------------------------------------------------------------------------------------------------------------
```

ASCII TABLE

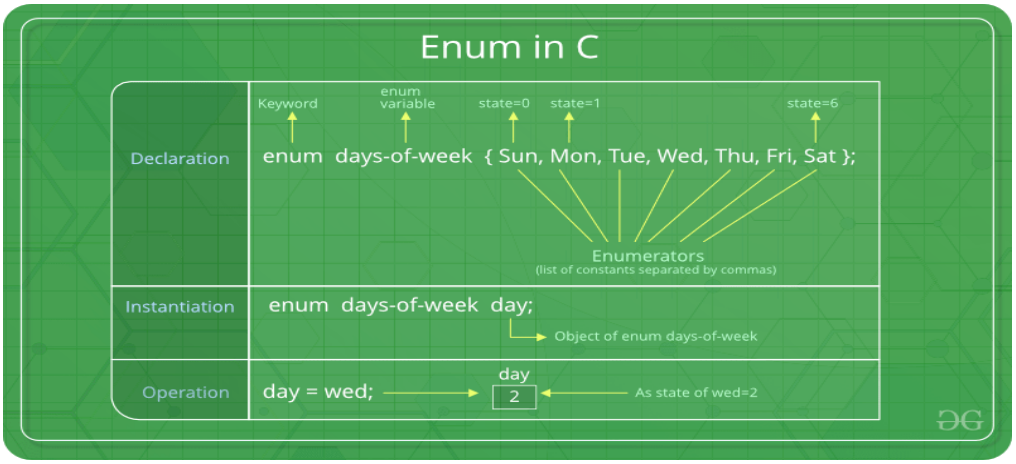
Dec	Char	Dec	Char	Dec	Char	Dec	Char
0	NUL (null)	32	SPACE	64	@	96	`
1	SOH (start of heading)	33	!	65	A	97	a
2	STX (start of text)	34	"	66	B	98	b
3	ETX (end of text)	35	#	67	C	99	c
4	EOT (end of transmission)	36	$	68	D	100	d
5	ENQ (enquiry)	37	%	69	E	101	e
6	ACK (acknowledge)	38	&	70	F	102	f
7	BEL (bell)	39	'	71	G	103	g
8	BS (backspace)	40	(	72	H	104	h
9	TAB (horizontal tab)	41	)	73	I	105	i
10	LF (NL line feed, new line)	42	*	74	J	106	j
11	VT (vertical tab)	43	+	75	K	107	k
12	FF (NP form feed, new page)	44	,	76	L	108	l
13	CR (carriage return)	45	-	77	M	109	m
14	SO (shift out)	46	.	78	N	110	n
15	SI (shift in)	47	/	79	O	111	o
16	DLE (data link escape)	48	0	80	P	112	p
17	DC1 (device control 1)	49	1	81	Q	113	q
18	DC2 (device control 2)	50	2	82	R	114	r
19	DC3 (device control 3)	51	3	83	S	115	s
20	DC4 (device control 4)	52	4	84	T	116	t
21	NAK (negative acknowledge)	53	5	85	U	117	u
22	SYN (synchronous idle)	54	6	86	V	118	v
23	ETB (end of trans. block)	55	7	87	W	119	w
24	CAN (cancel)	56	8	88	X	120	x
25	EM (end of medium)	57	9	89	Y	121	y
26	SUB (substitute)	58	:	90	Z	122	z
27	ESC (escape)	59	;	91	[	123	{
28	FS (file separator)	60	<	92	\	124	|
29	GS (group separator)	61	=	93	]	125	}
30	RS (record separator)	62	>	94	^	126	~
31	US (unit separator)	63	?	95	_	127	DEL

Header File in C/old C++ vs ANSI C++/New C++

Old style	New style
<assert.h>	<cassert>
<ctype.h>	<cctype>
<float.h>	<cfloat>
<limits.h>	<climits>
<math.h>	<cmath>
<stdio.h>	<cstdio>
<stdlib.h>	<cstdlib>
<string.h>	<cstring>
<time.h>	<ctime>

ENUMERATION

Enumeration (or enum) is a user defined data type in C/C++ .
It is mainly used to assign names to integral constants, the names make a program easy to read and maintain.
The keyword ‘enum’ is used to declare new enumeration types in C and C++.



1. Two enum names can have same value. For example, in the following C program both 'Failed' and 'Freezed' have same value 0.
enum State {Working = 1, Failed = 0, Freezed = 0};

1. If we do not explicitly assign values to enum names, the compiler by default assigns values starting from 0. For example, in the following C program, sunday gets value 0, monday gets 1, and so on.
enum day {sunday, monday, tuesday, wednesday, thursday, friday, saturday};

2. We can assign values to some name in any order. All unassigned names get value as value of previous name plus one.
enum day {sunday = 1, monday, tuesday = 5,wednesday, thursday = 10, friday, saturday};

3. The value assigned to enum names must be some integeral constant, i.e., the value must be in range from minimum possible integer value to maximum possible integer value.

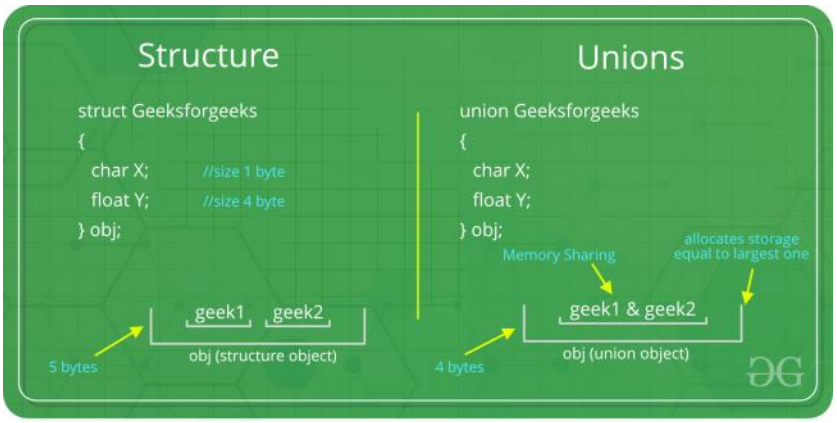
1. All enum constants must be unique in their scope. For example, the following program fails in compilation.

enum state {working, failed};
enum result {failed, passed};

2. Enum { X , Y , Z } ;
   is equivalent to
   Const X = 0;
   Const Y = 1;
   Const Z = 2;

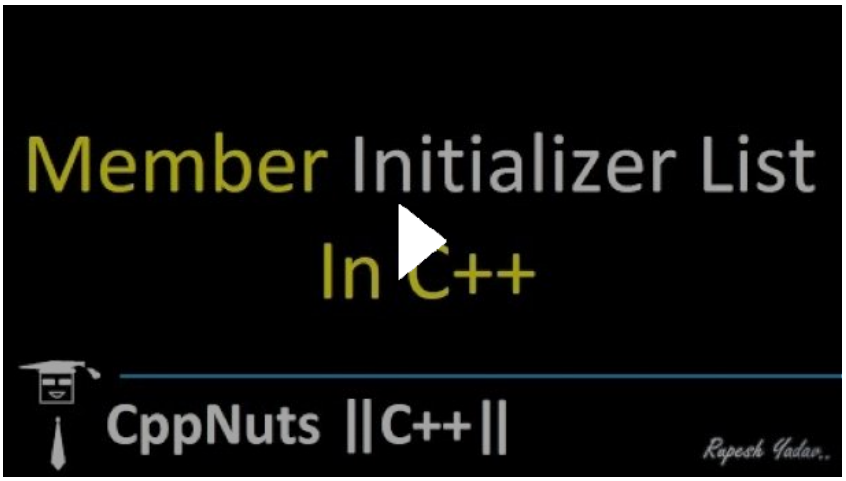
UNION

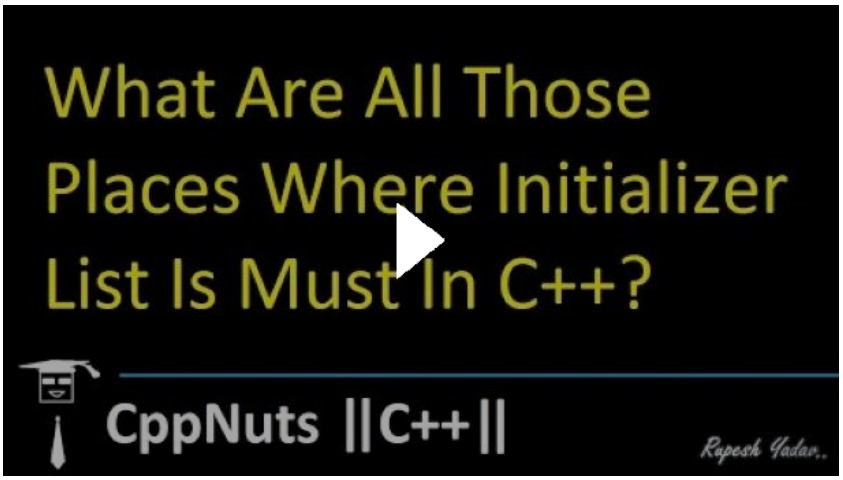
Union is a user-defined datatype. All the members of union share same memory location. Size of union is decided by the size of largest member of union. If you want to use same memory location for two or more members, union is the best for that. Unions are similar to structures. Union variables are created in same manner as structure variables. The keyword “union” is used to define unions in C/C++ language. Like structures, we can have pointers to unions and can access members using the arrow operator (->).Unions can be useful in many situations where we want to use the same memory for two or more members.



Initializer List

https://www.geeksforgeeks.org/when-do-we-use-initializer-list-in-c/Initializer List In C++





Tokens

Tokens : Smallest individual units in a program are known as tokens.
Example) keywords, identifiers, constants, strings and operators.

1. ***Keywords***: They are *explicitly reserved identifiers* which cannot be used as names for program variables or functions.

Keywords

auto	double	int	struct	asm	private
break	else	long	switch	catch	public
case	enum	register	typedef	class	protected
char	extern	return	union	delete	template
const	float	short	unsigned	friend	this
continue	for	signed	void	inline	throw
default	goto	sizeof	volatile	new	try
do	if	static	while	operator	virtual

Detailed List of Keywords:

alignas (since C++11) alignof (since C++11) and and_eq asm atomic_cancel (TM TS) atomic_commit (TM TS) atomic_noexcept (TM TS) auto(1) bitand bitor bool break case catch char char8_t (since C++20) char16_t (since C++11) char32_t (since C++11) class(1) compl concept (since C++20) const consteval (since C++20) constexpr (since C++11) constinit (since C++20) const_cast continue co_await (since C++20) co_return (since C++20) co_yield (since C++20) decltype (since C++11)	default(1) delete(1) do double dynamic_cast else enum explicit export(1)(3) extern(1) false float for friend goto if inline(1) int long mutable(1) namespace new noexcept (since C++11) not not_eq nullptr (since C++11) operator or or_eq private protected public reflexpr (reflection TS)	register(2) reinterpret_cast requires (since C++20) return short signed sizeof(1) static static_assert (since C++11) static_cast struct(1) switch synchronized (TM TS) template this thread_local (since C++11) throw true try typedef typeid typename union unsigned using(1) virtual void volatile wchar_t while xor xor_eq

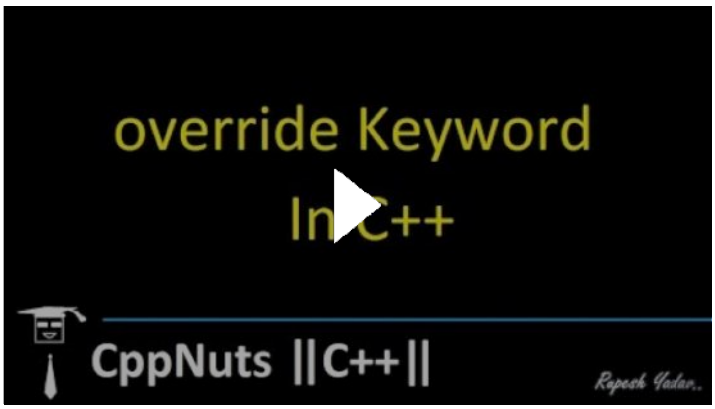
Some New Keywords in C++11:

- **Auto** => used for generic programming (Abdul Bari Video)

- **Final** => used to restrict a class to get inherited (add keyword final after base className), thus preventing inheritance. Also can be used to make a function which cannot be overriden(add keyword final after base member function and *also make it virtual*)
(Abdul Bari Video)

- **Override** => (Optional) Add override keyword after agument list and before { brace to declare a function which has overrided a base class function. Testing becomes easy (as explicitly it is shown that it is overrided function) and compile time check can be performed so that error in future(run time) can be avoided.

  Note: Difference between new, override and virtual:
  virtual: indicates that a method may be overriden by an inheritor
  override: overrides the functionality of a virtual method in a base class, providing different functionality.
  new: hides the original method (which doesn't have to be virtual), providing different functionality. This should only be used where it is absolutely necessary.



- **Explicit** => used to avoid implicit call to the constructor(passing one parameter to the constructor).
  Article: https://www.geeksforgeeks.org/g-fact-93/

The explicit Keyword

The **explicit** keyword is used to declare class constructors to be "explicit" constructors. We have seen earlier, while discussing constructors, that any constructor called with one argument performs *implicit conversion* in which the type received by the constructor is converted to an object of the class in which the constructor is defined. Since the conversion is automatic, we need not apply any casting. In case, we do not want such automatic conversion to take place, we may do so by declaring the one-argument constructor as explicit as shown below:

```
class ABC
{
      int m;
  public:
      explicit ABC (int i)      // constructor
      {
            m = i;
      }
      // ..............
      // ..............
};
```

Here, objects of ABC class can be created using only the following form:

```
ABC abc1(100);
```

The automatic conversion form

```
ABC abc1 = 100;
```

is not allowed and illegal. Remember, this form is permitted when the keyword **explicit** is not applied to the conversion.

- **Mutable** => to allow a variable to be modified which is defined inside a *constant object or constant function*. Append mutable keyword before datatype of variable.

The mutable Keyword

We know that a class object or a member function may be declared as **const** thus making their member data not modifiable. However, a situation may arise where we want to create a **const** object (or function) but we would like to modify a particular data item only. In such situations we can make that particular data item modifiable by declaring the item as **mutable**. Example:

```
mutable int m;
```

Although a function(or class) that contains **m** is declared **const**, the value of **m** may be modified. Program 16.2 demonstrates the use of a **mutable** member.

2. ***Identifiers:*** They refer to names of variables , functions, arrays, classes and other user defined data types created by programmer.

- Note: C language has limit on length of identifier name as it recognizes only first 32 characters but there is no limit in C++ language.
- Note: All variables must be declared in starting of scope in C language but in C++, they can be declared anywhere in the scope even just before they are used.
- Note: **Dynamic Initialization** of variables is possible only in C++ language and not in C language. Dynamic Initialization means declaring the variable first and then initializing it later (in different statement).

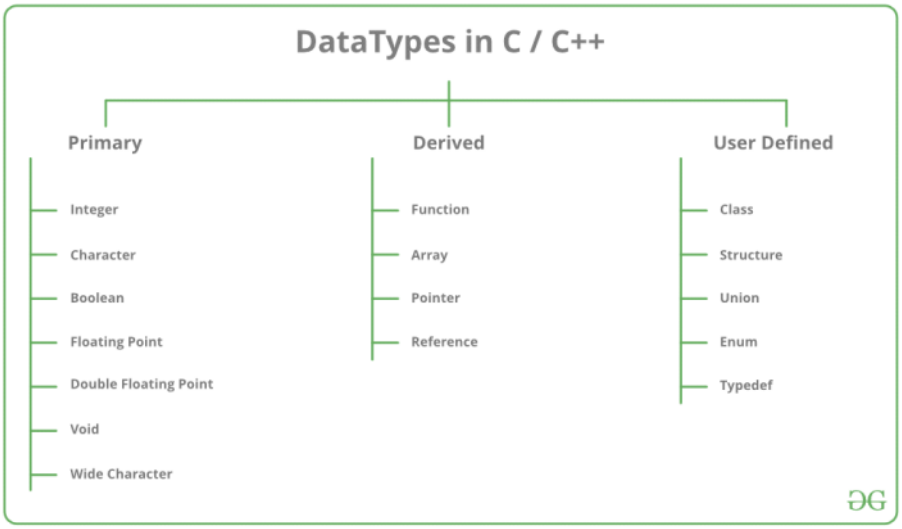
3. ***Constants:*** They refer to the fixed values that do not change during the execution of program.

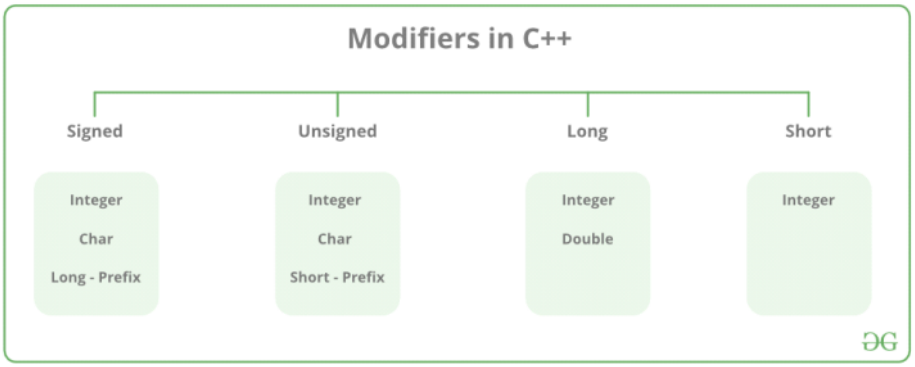
   Example)
   123 => Decimal Integer
   12.34 => Floating point Integer
   037 => Octal Integer
   OX2 => Hexadecimal Integer
   "C++" => String literal Constant
   'A' => Character Constant
   \n => Backlash Character Constants
   L 'ab' => Wide-Character Constant
   { Begins with L and stored in '***wchar_t***' datatype where characters sets cannot fit into single byte.
   wchar_t holds 16-bit characters which are used to represent character set of languages having more than 255 characters such as Japanese.}

   Note: There are **symbolic constants** using either qualifier '*const*' or set of integer constants using '*enum*';

4. ***Data Type:***

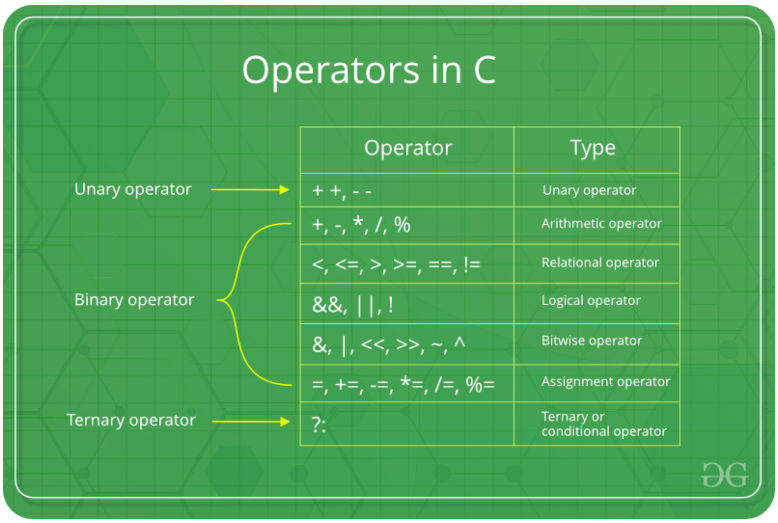
   Note: sizeof('A') = sizeof(int) in C but sizeof(char) in C++





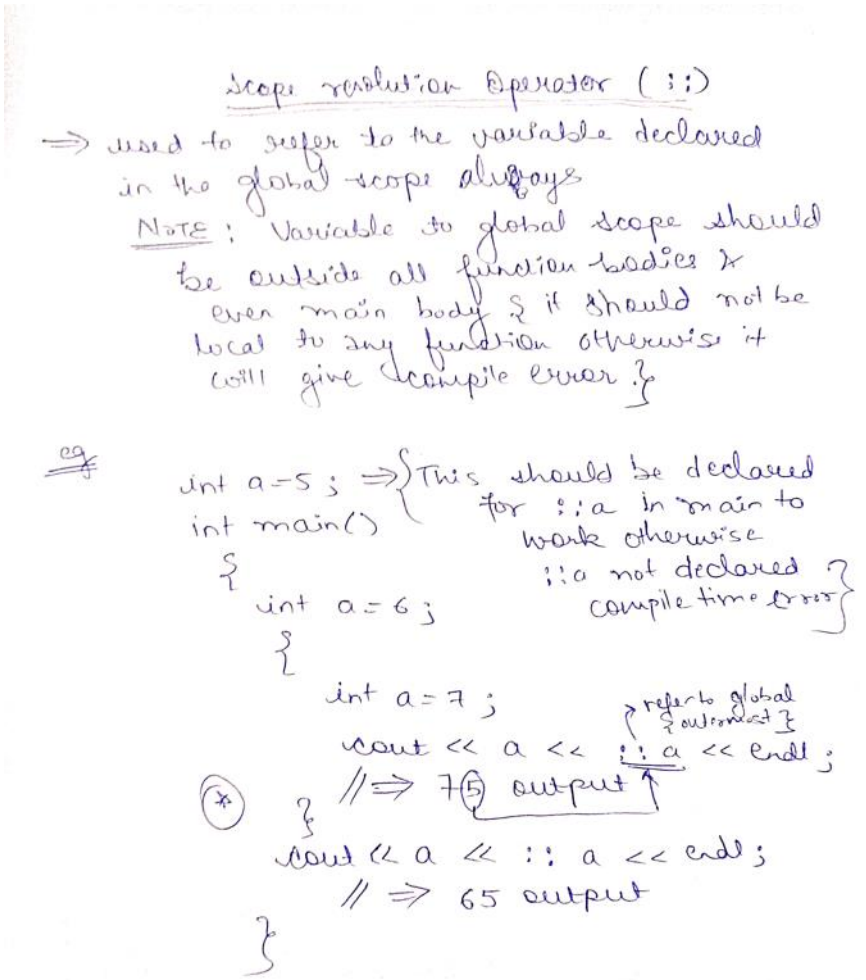
5. ***Operators***

Refer: https://www.geeksforgeeks.org/operators-c-c/



Note: Following operators are only valid in C++ but not in C:

1. Insertion Operator (<<)
2. Extraction Operator (>>)
3. Scope Resolution Operator (::)



4. Pointer to Member declarator (::*)
5. Pointer to Member operators (->* , .*)

   Note: Collectively Pointer to Member declarator and operators are known as **Member Dereferencing Operators**.

6. Memory Allocation operator (new)

   It allocates sufficient memory to hold a data object of given datatype and returns the address of the object.

   *Declaration: DataType *PointerVariable = new DataType;*

   *Declaration with Initialization: DataType *PointerVariable = new DataType(value);*

   Declaration of Array: DataType *PointerVariable = new DataType[size] ;

   Note: For Declaring multi-dimensional array, first dimension may be a variable whose value is supplied at runtime but all other **must** be constants. Example:

   ```
   int *arr = new int[3][4]; // Valid
   int *arr = new int[][4]; // Invalid
   int *arr = new int[3][]; // Invalid
   Int *arr = new int[][]; // Invalid
   Int *arr = new int[n][4]; // Valid
   ```

Int *arr = new int[3][m]; // Invalid

Note: If sufficient memory is not available for allocation, new returns a null pointer. Therefore it is a good practice to check if pointer points to NULL before using it.

Note: Advantages of '***new***' over '***malloc()***':

- It automatically computes the size of data object. We need not use '*sizeof*' operator.
- It automatically returns the correct pointer type. We need not use *Explicit Type Casting*.
- It is possible to initialize the object while creating the memory space (in the same line only).
- new & delete are operators (rather than functions) and thus *operator overloading* is possible.

7. Memory Release operator (delete)

   It is used to destroy a data or object when no longer needed and release the memory space for reuse.

   Syntax: delete PointerVariable; // For Datatypes other than arrays.

   delete[size] PointerVariable; // For Arrays

   delete[] PointerVariable; // For Arrays without giving size

   Note: Collectively *new* & *delete* are known as **Memory Management Operators.**

   Since these operators manipulate memory on the free store, they are also known as **Free Store Operators**.

1. Line Feed operator (endl)
2. Field Width operator (setw)

Note: endl, setprecision and *setw* are examples of **Manipulators**.

They are operators that are used to format the data display.

8. Typeid Operator (in C++11)

   Used to obtain types of unknown objects, such as their class name at runtime known as *RunTime Type Information*(RTTI).

   Eg) char *objectType = typeid(object).name();

   Will assign type of "object" to the character array objectType. The object may be of primitive or user defined type.

   Note: We must include <**typeinfo**> headerfile to use typeid operator.

 # Operator Precedence Chart

Level	Operators	Description	Associativity
15	() [] -> . ++ --	Function Call Array Subscript Member Selectors Postfix Increment/Decrement	Left to Right
14	++ -- + - ! ~ (type) * & sizeof	Prefix Increment / Decrement Unary plus / minus Logical negation / bitwise complement Casting Dereferencing Address of Find size in bytes	Right to Left
13	* / %	Multiplication Division Modulo	Left to Right
12	+ -	Addition / Subtraction	Left to Right
11	>> <<	Bitwise Right Shift Bitwise Left Shift	Left to Right
10	< <= > >=	Relational Less Than / Less than Equal To Relational Greater / Greater than Equal To	Left to Right
9	== !=	Equality Inequality	Left to Right
8	&	Bitwise AND	Left to Right
7	^	Bitwise XOR	Left to Right
6	|	Bitwise OR	Left to Right
5	&&	Logical AND	Left to Right
4	||	Logical OR	Left to Right
3	?:	Conditional Operator	Right to Left
2	= += -= *= /= %= &= ^= |= <<= >>=	Assignment Operators	Right to Left
1	,	Comma Operator	Left to Right

#operator keywords

Table 16.1 *Operator keywords*

Operator	*Operator keyword*	*Description*
&&	and	logical AND
||	or	logical OR
!	not	logical NOT
!=	not_eq	inequality
&	bitand	bitwise AND
|	bitor	bitwise inclusive OR
^	xor	bitwise exclusive OR
~	compl	bitwise complement
&=	and_eq	bitwise AND assignment
|=	or_eq	bitwise inclusive OR assignment
^=	xor_eq	bitwise exclusive OR assignment

Storage Classes

Storage Class	Keyword	Default Value	Storage	Scope	Life
Automatic	**auto**	Garbage	RAM	Limited to the block in which it is declared	Till the execution of the block in which it is declared
Register	**register**	Garbage	register	Same	same
Static	**static**	0 (zero)	RAM	Same	Till the end of program
External	**extern**	0 (zero)	RAM	Global	Same

6. ***Expressions***

It is a combination of operators, constants and variables arranged as per the rules of language.
It may also include function calls which return values.

Types:

1. *Constant Expression: Consists of only constant values like 15 , 20 + 5/2.0 , 'x' , etc.*
2. *Integral Expression: which produce integer results after all automatic and explicit type conversions.*
   *Eg) m , m + n - 2 , m * 'x' , 5 + int(2.0) , etc where m & n are integer variables*
3. ***Float Expressions:** which produce floating-point results after all conversions.*
   Eg) x + y, 5 + float(10) , 10.75 , etc. where x & y are floating point variables
4. ***Pointer Expressions:** which produce address values like &m , ptr , ptr + 1 , "xyz"*
5. Relational or Logical Expressions: which produce boolean result (either true or false) like a > b && x == 10
6. ***Bitwise Expressions:*** manipulate data at bit level like left shift or right shift.

Manipulators

Manipulators are helping functions that can modify the input/output stream. It does not mean that we change the value of a variable, it only modifies the I/O stream using insertion (<<) and extraction (>>) operators.

For example, if we want to print the hexadecimal value of 100 then we can print it as: cout<<setbase(16)<<100;

Types of Manipulators

There are various types of manipulators:

1. **Manipulators without arguments**: The most important manipulators defined by the **IOStream library** are provided below.

- **endl**: It is defined in ostream. It is used to enter a new line and after entering a new line it flushes (i.e. it forces all the output written on the screen or in the file) the output stream.
- **ws**: It is defined in istream and is used to ignore the whitespaces in the string sequence.
- **ends**: It is also defined in ostream and it inserts a null character into the output stream. It typically works with std::ostrstream, when the associated output buffer needs to be null-terminated to be processed as a C string.
- **flush**: It is also defined in ostream and it flushes the output stream, i.e. it forces all the output written on the screen or in the file. Without flush, the output would be the same, but may not appear in real-time.

2. **Manipulators with Arguments:** Some of the manipulators are used with the argument like setw (20), setfill ('*'), and many more. These all are defined in the header file. If we want to use these manipulators then we must include this header file in our program.
   For Example, you can use following manipulators to set minimum width and fill the empty space with any character you want: std::cout << std::setw (6) << std::setfill ('*');

   Some important manipulators in *<iomanip>* are:

   a. **setw (val):** It is used to set the field width in output operations.
   b. **setfill (c):** It is used to fill the character 'c' on output stream.
   c. **setprecision (val):** It sets val as the new value for the precision of floating-point values.
   d. **setbase(val):** It is used to set the numeric base value for numeric values.
   e. **setiosflags(flag):** It is used to set the format flags specified by parameter mask.
   f. **resetiosflags(m):** It is used to reset the format flags specified by parameter mask.
   g. **Some important manipulators in <ios> are:**
   h. **showpos:** It forces to show a positive sign on positive numbers.
   i. **noshowpos:** It forces not to write a positive sign on positive numbers.
   j. **showbase:** It indicates the numeric base of numeric values.
   k. **uppercase:** It forces uppercase letters for numeric values.
   l. **nouppercase:** It forces lowercase letters for numeric values.
   m. **fixed:** It uses decimal notation for floating-point values.
   n. **scientific:** It uses scientific floating-point notation.
   o. **hex:** Read and write hexadecimal values for integers and it works same as the setbase(16).
   p. **dec:** Read and write decimal values for integers i.e. setbase(10).
   q. **oct:** Read and write octal values for integers i.e. setbase(10).
   r. **left:** It adjusts output to the left.
   s. **right:** It adjusts output to the right.

STRUCTURE DIFFERENCE IN C & C++

1. **For declaring object variable of structure**

in C
struct student { BODY } ;
struct student S1;

in C++
struct student { BODY } ;
_ _ _ student S1 ;
{ We can omit struct in declaring object of struct }

2. **Member functions inside structure:** Structures in C cannot have member functions inside structure but Structures in C++ can have member functions along with data members**.** Member functions in C++ structure are by default public.

3. **Constructor creation in structure:** Structures in C cannot have constructor inside structure but Structures in C++ can have Constructor creation.

4. **Direct Initialization:** Direct Initialization without constructor is allowed in C++ but not in C. struct student { int rollno = 1; } => Valid in C++

4. **C structure cannot have static data members but C++ structure can have.**

5. **Size of empty structure**
   struct Empty { } ;
   cout<<sizeof(Empty); => 1 answer in C++
   printf("%d",sizeof(Empty)); => 0 answer in C

6. **C structure do not have access modifiers like private, protected**

C => Data members are public by default and cannot be changed thus OOPS is not implemented.
C++ => Data members and member functions in C++ structure are also public by default but their access specifier can be changed.

STRUCTURE IN C++ vs CLASS IN C++

Class: User-defined blueprint from which objects are created. It consists of methods or set of instructions that are to be performed on the objects.

Structure: A structure is basically a user-defined collection of variables which are of different data types.

Differences in Class & Structure

1. **Security:** A Structure is not secure(by default) and cannot hide its implementation details from the end user while a class is secure and can hide its programming and designing details.

2. **Default Access Specifier :** Members of a class are private by default and members of a struct are public by default.

3. **Default Visibility Mode** : Default visibility mode while inheriting base structure in child structure is public while it is private in class.

Similarities in Class & Structure

1. Both can have constructors, methods, properties, fields, constants, enumerations, events, and event handlers.
2. Both structures and classes can implement interfaces to use multiple-inheritance in code.
3. Both structures and classes can have constructors with parameter.
4. Both structures and classes can have delegates and events.

Note: Though structure and class in C++ differ only by default access specifier and visibility mode, we usually:

- use **struct** for plain-old-data structures without any class-like features;
- use **class** when you make use of features such as private or protected members, non-default constructors and operators, etc.

Benefits of function

- **Modularization**: Dividing a complex program into simpler blocks
- Easy to read, debug and modify code (**Easy Maintenance** & **Readability**)
- Avoids repetition of code (**Reusability**)
- Better memory utilization
- Function Overloading

Note: Anonymous function or Lambda Expression in Abdul Bari Video.
Note: Ellipsis(…) in Abdul Bari Video.

Main Function

*Prototype: int main(int argc, char * argv[]) or simply int main()*

- main is not a keyword in C/C++. It is a function name which is not predefined like keywords. (Although it is predeclared like function prototype in preprocessor and hence it cannot be used as an identifier name) .Thus it is itself also an identifier.
  Note: Other such predefined identifiers are *cin, cout, endl, include, iostream, INT_MAX, INT_MIN, std, string, NULL*, etc.
- int main ( ) without any return statement is valid . Zero (0) is returned by default { normal termination }. However it is good practice to write return 0 statement as it tells compiler it is a normal termination and program ran successfully and non - zero value tells there is abnormal termination.
- void main ( ) or main ( ) is invalid in both C & C++. Although some compilers (like Turbo C++) may run without error.

Function(void) vs Function()

In C

Function call without any arguments i.e. function() takes it as functional calls with *unspecified* number of arguments. While function(void) strictly means no arguments. Thus it is good practice to use void when there are no arguments.

In C++

Both statements are same in C++ which means function call with no arguments.

Function Prototyping

It describes the function interface to the compiler by giving details such as number and type of arguments and the return type. *It is essential in C++ but optional in C.*

Prototype: Return-Type Function-Name (argument list);

Note: In **Function Declaration** names of arguments are dummy variables and therefore they are optional. Eg) float volume(int , int , int) is valid only in function declaration not in function prototype.

Inline Functions

Every time a function is called, it takes a lot of extra time in executing a series of instructions for tasks such as jumping to the function, saving registers, pushing arguments into the stack, and returning to the calling function. When a function is small, a substantial percentage of execution time may be spent in such overheads.

To eliminate cost of calls to small functions inline functions are used. It is a function that is expanded in line in which it is invoked i.e. compiler replaces the function call with the corresponding function code. To make a function inline, we need to prefix the keyword ***inline*** to the function definition. All inline functions must be defined before they are called.
Inline function makes a program runs faster because the overhead of a function call and return is eliminated.

inline keyword merely sends a request, not a command, to the compiler.
The compiler may ignore this request if the function definition is too long or too complicated and compile the function as a normal function.

Some of the situations where inline expansion may not work are:

- For functions returning values, if a loop, a switch, or a goto exists.
- For functions not returning values, if a return statement exists.
- If functions contain static variables.
- If inline functions are recursive.

Function Overloading

"Use of same function name to create functions that perform a variety of different tasks is known as function overloading."
Functions overloaded (with same function name) must have difference in argument list(number of arguments or type of arguments).
Function overloading finds for best match which must be unique otherwise it will cause ambiguity.

Q) Which functions cannot be overloaded ?

1. Functions with same argument list but with *different return type* are not overloaded functions and hence will give redefinition of function compile time error.
2. Member function declarations with the same name and the name parameter-type-list cannot be overloaded if any of them is a static member function declaration.
3. Parameter declarations that differ only in a pointer * versus an array [] are equivalent, hence cannot be overloaded. For eg) `int fun(int *ptr);` and `int fun(int ptr[]);` are same and will give re-declaration error.
4. Parameter declarations that differ only in that one is a function type and the other is a pointer to

the same function type are equivalent. For Eg) `void h(int ());` and `void h(int (*)());` will give re-declaration error.

5. Parameter declarations that differ only in the presence or absence of *const* and/or *volatile* are equivalent. For eg) `int f ( int x)` And `int f ( const int x)` Will give re-declaration error.

Steps of Selection of function on function call:

1. **Exact unique match** (exactly same number and type of arguments)
2. If exact match is not found, compiler uses **integral promotions** to the actual arguments:
   (char to **int**) and (**float** to **double**) to find a match.
3. If integral promotions don't work, compiler tries to use **built-in conversions** to actual arguments and finds a match. However, if found multiple matches, it causes ambiguity and compiler gives error.
4. If built in type conversions don't work, compiler try **user-defined conversions** in combination with integral promotions and built in conversions.
   If still exact match is not found , compiler gives function not found error.

Note: Function with 1 default argument will cause ambiguity to function with no arguments on function call with no arguments passed.

Exception Handling

Types of Errors:

1. Syntax Error/Compile Time Error: Caused due to poor understanding of language used.
   => Caught by Compiler
2. *Logical Error or Symantic Error*: Caused due to poor understanding of problem statement and improper logic.
   => Caught by Debugger
3. *RunTime Error*: Caused by bad input by user or non-availability of appropriate resources. If not caught using *exception handling* (try-throw-catch), then will cause program to terminate abnormaly or show unexpected behavior at run-time.

   Must Watch: Below video & Abdul Bari videos of Exception Handling.

[Lecture 27 Exception Handling in C++ Hindi]

Exception is any abnormal behavior, run-time error or an off-beat situation.
Eg) Division by zero, access to an array outside of its bounds, running out of memory or disk space. Program should be ready to handle it with appropriate response. Such a program which is able to handle every exception appropriately is said to be ***robust*** in nature.

Q) Difference between Error and Exception?
Exceptions are also known as run-time errors. Hence an Exception is also an error. But it's not detected during compilation phase unlike Syntactic errors and also they don't execute normally during run time, unlike semantic errors. These errors are detected only during runtime and prevent the normal execution of a program.

Note: Exceptions are of two kinds, namely, ***synchronous exceptions*** and ***asynchronous exceptions***. Errors such as "out of range index" and "over-flow" belong to the synchronous type exceptions. The errors that are caused by events beyond the control of the program (such as keyboard interrupts) are called asynchronous exceptions. The proposed handling mechanism in ***C++*** is designed to ***handle only synchronous exceptions***.

The built-in mechanism to handle exceptions given by C++ is known as ***exception handling***.
Using exception handling, we can easily manage and respond to run-time errors.
Exception handling mechanism provide a way to transfer control from one part of program to another.
This makes it easy to separate the error handling code from the code written to handle the actual functionality of the program.

Syntax

```
try{
}
catch(type1 arg) {
}
catch(type2 arg) {
}
...
catch(typeN arg) {
}
```

C++ exception handling is built upon three keywords: ***try***, ***catch*** and ***throw.***

- **try**:
  A block of code which may cause an exception is typically placed inside the try block.
  Every try catch **must** have a corresponding catch block. A single try block can have multiple (alteast 1) catch blocks.
  If an exception occurs, it is thrown from the try block.

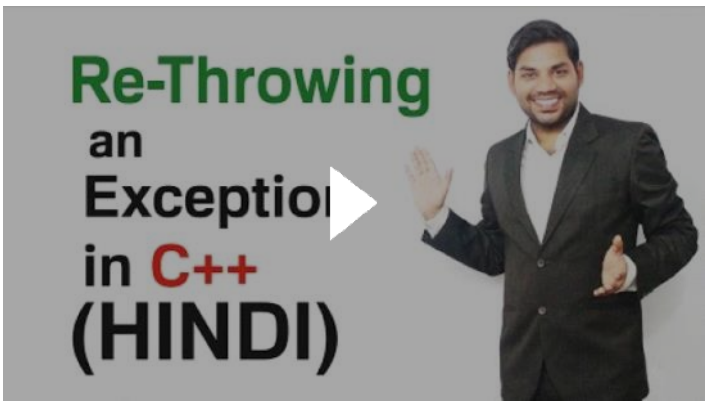
- **catch** :
  this block catches the exception thrown from the try block.
  Code to process and handle the exception is written inside this catch block.
  When an exception is caught, arg will receive its value.
  If you don't need access to the exception itself, specify only type in the catch clause — arg is optional.
  Any type of data can be caught, including classes that you create.

- **throw** :
  A program throws an exception when a problem shows up within the try block.
  The general form of the throw statement is: *throw exception*;
  Throw must be executed either within the try block proper or from any function that the code within the try block calls.
  The point at which throw is executed is known as *throw point*.

Q) : Why try-throw-catch should be used instead of if-else for exception handling?
A) : There may be use of functions in which logic of program is written. Now exceptions might occur

inside function body and function need to report exception back to the function (main) calling it. We will be forced to *return* the exception(in the same datatype as the return type) or handle the exception inside function only if using if-else statements, but if using try-catch-throw, we can wrap function call statement inside try block and *throw* any type of exception (other than return type of function) from function body instead of returning it. Thus it is better to use try-catch-throw instead of if-else for exception handling.

Rethrowing Exception



Namespace

Preprocessor Directive

Preprocessor directives are lines included in the code of our programs that are not program statements but directives for the preprocessor. These lines are always preceded by a hash sign (#). The preprocessor is executed before the actual compilation of code begins, therefore the preprocessor digests all these directives before any code is generated by the statements.

These preprocessor directives extend only across a single line of code. As soon as a newline character is found, the preprocessor directive is considered to end. No semicolon (;) is expected at the end of a preprocessor directive.

macro definitions (#define, #undef)

To define preprocessor macros we can use #define. Its format is:
#define identifier replacement

When the preprocessor encounters this directive, it replaces any occurrence of identifier in the rest of the code by replacement. This replacement can be an expression, a statement, a block, function macros or simply anything. The preprocessor does not understand C++, it simply replaces any occurrence of identifier by replacement.

Source file inclusion (#include)

When the preprocessor finds an #include directive it replaces it by the entire content of the specified file.
There are two ways to specify a file to be included:

1. *#include "file"*
2. *#include <file>*

The only difference between both expressions is the places (directories) where the compiler is going to look for the file. In the first case where the file name is specified between double-quotes, the file is searched first in the same directory that includes the file containing the directive. In case that it is not there, the compiler searches the file in the default directories where it is configured to look for the standard header files.
If the file name is enclosed between angle-brackets <> the file is searched directly where the compiler is configured to look for the standard header files. Therefore, standard header files are usually included in angle-brackets, while other specific header files are included using quotes.

Conditional inclusions (#ifdef, #ifndef, #if, #endif, #else and #elif)

These directives allow to include or discard part of the code of a program if a certain condition is met.

#ifdef allows a section of a program to be compiled only if the macro that is specified as the parameter has been defined, no matter which its value is.

#ifndef serves for the exact opposite: the code between #ifndef and #endif directives is only compiled if the specified identifier has not been previously defined.

#ifndef serves for the exact opposite: the code between #ifndef and #endif directives is only compiled if the specified identifier has not been previously defined.

Must Watch These Videos (Not in Book or Notes) + ***Abdul Bari Header File Video***

Lecture 29 namespace in C++ Part 1 of 2 Hindi

Lecture 29 namespace in C++ Part 2 of 2 Hindi

Namespace is a container for identifiers. It puts the names of its members in a distinct space so that they don't conflict with the names in other namespaces or in global namespace.

Defining a namespace

We can define our own namespace whose syntax is similar to defining a class. Although there is no termination semicolon '*;*' after closing brace '*}*'.

```
namespace namespace_name
{
        // Declaration of
        // variables, functions, classes, etc.
}
```

We can ***declare*** variables, functions, classes ,etc in the namespace and define them outside namespace using scope resolution operator (namespace_name::) or define them itself in the

namespace.

Example:

```
namespace TestSpace
{
      int m;
      void display(int n)
     {
           cout << n;
     }
}          // No semicolon here
```

Here, the variable **m** and the function **display** are inside the scope defined by the **TestSpace** namespace. If we want to assign a value to m, we must use the scope resolution operator as shown below.

```
TestSpace::m = 100;
```

Note that **m** is qualified using the namespace name.

This approach becomes cumbersome if the members of a namespace are frequently used. In such cases, we can use a **using** directive to simplify their access. This can be done in two ways:

```
using namespace namespace_name;        // using directive

using namespace_name::member_name;          // using declaration
```

In the first form, all the members declared within the specified namespace may be accessed without using qualification. In the second form, we can access only the specified member in the program. Example:

```
using namespace TestSpace;
m = 100;          // OK
display(200);     // OK

using TestSpace::m;
m = 100;          // OK
display(200);     // Not ok, display not visible
```

Data File Handling in C

C++ uses the concept of ***stream*** and ***stream classes*** to implement its I/O operations with the console and the disk files. C++ supports all of C's rich set of I/O functions. We can use any of them in the C++ programs, but they are not commonly used because:

- I/O methods in C++ support the concepts of OOP.
- I/O methods in C cannot handle the user- defined data types such as class objects.

File Opening Modes

Mode	Meaning	Description
r	Read	Only reading possible. Not create file if not exist
w	Write	Only writing possible. Create file if not exist otherwise erase the old content of file and open as a blank file
a	Append	Only writing possible. Create file if not exist, otherwise open file and write from the end of file (do not erase the old content)
r+	Reading + Writing	R & W possible. Create file if not exist. Overwriting existing data. Used for modifying content
w+	Reading + Writing	R & W possible. Create file if not exist. Erase old content.
a+	Reading + Appending	R & W possible. Create file if not exist. Append content at the end of file

```
#include <stdio.h>
int main()
{
// FILE is a non-primitive data type in stdio.h Header file.
FILE *fp;
// fp is a pointer of type FILE which will point to address of a file when it is loaded in buffer.
fp = fopen("txtfile.txt","w");
/* fopen returns a address of type FILE. It takes two arguments
1. name of text/binary file
2. file opening mode ( a , w , r , a+ , w+ , r+ , ab , wb , rb , ab+ , wb+ , rb+ )
*/
// fopen() will return NULL if FILE is not found in case of r or r+ opening mode.
// Otherwise it will return NULL if it cannot open.
if(fp == NULL)
{
printf("File cannot open or not found");
exit(0);
}
// 1. fputc() => Write character by character in file.
char str[100];
printf("Enter String : ");
gets(str);
for(int i=0;i<strlen(str);i++)
fputc(str[i],fp);
fclose(fp); // fclose() closes opened file => remove the file from buffer in RAM. Store it only in Hard disk.
// 2. fgetc => Read character by character from file.
fp = fopen("txtfile.txt","r");
char ch = fgetc(fp);
```

```
while(!feof(fp)) // feof() returns true if end of file is reached else returns false.
{
printf("%c",ch);
ch = fgetc(fp);
}
printf("\n");
fclose(fp);
// 3. fputs() => Write line by line on file.
fp = fopen("txtfile.txt","w");
fputs(str,fp);
fclose(fp);
// 4. fgets() => Read line by line ( or string by string ) from file.
fp = fopen("txtfile.txt","r");
char s[100];
while(fgets(s,99,fp) != NULL) // fgets() returns false if no more string in file remains otherwise copies it in s;
{
printf("%s",s);
}
printf("\n");
fclose(fp);
// 5. fwrite() => Write data from variable( of any data type including UDT struct ) on file.
// => used for binary files.
fp = fopen("binfile.dat","wb");
struct person
{
int age;
int rollno;
}p1,p2;
printf("\n Enter age and roll no : ");
scanf("%d",&p1.age);
scanf("%d",&p1.rollno);
fwrite(&p1,sizeof(p1),1,fp);
// four arguments : add(variable) , sizeof(variable) , no of records to write , file pointer
fclose(fp);
// 6. fread() => Read data from file using a variable.
fp = fopen("binfile.dat","rb");
fread(&p2,sizeof(p2),1,fp); // fread() returns false if no more record is to be read.
printf("\n%d %d\n",p2.age,p2.rollno);
fclose(fp);
// 7. fprintf() => Write the output from screen to the file and dont't display on screen.
fp = fopen("sum.txt","w");
int a,b;
printf("\n Enter two numbers : ");
scanf("%d %d", &a , &b );
fprintf(fp,"sum of %d and %d is %d ",a,b,a+b);
// two arguments : file pointer and the statement of printf()
```

```
fclose(fp);
// 8. fscanf => Read from file and store it directly in variables.
fp = fopen("sum.txt","r");
fscanf(fp,"%d%d",&a,&b);
printf("a= %d b = %d\n",a,b);
fclose(fp);
return 0;
}
```

C++ Streams

The I/O system in C++ is designed to work with a wide variety of devices including terminals, disks, and tape drives.
Although each device is very different, the I/O system supplies an interface to the programmer that is independent of the actual device being used/accessed. This interface b/w the program and the input/output device is known as ***stream***.
A stream is a sequence of bytes. It acts either as a *source* from which the input data can be obtained or as a *destination* to which the output data can be sent.

The source stream that provides data to the program is called the ***input stream*** and the destination stream that receives output from the program is called the ***output stream***.
The data in the input stream can come from the keyboard or any other storage device. Similarly, the data in the output stream can go to the screen or any other storage device.
In other words, a program extracts the bytes from an input stream and inserts bytes into an output stream.

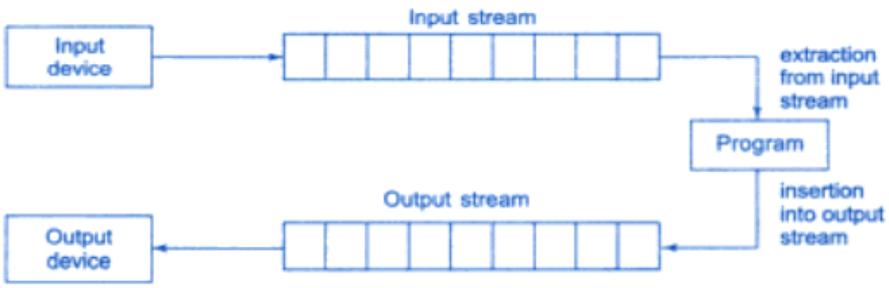


Fig. 10.1 ⇔ *Data streams*

C++ contains several pre-defined streams that are automatically opened when a program begins its execution.
These include ***cin*** and ***cout.***

cin represents the input stream connected to the standard input device (usually the keyboard) and *cout* represents the output stream connected to the standard output device (usually the screen).

C++ Stream Classes

The C++ I/O system contains a hierarchy of classes that are used to define various streams to deal with the console and disk files. These classes are called ***stream classes***.

Figure shows the hierarchy of the stream classes used for input and output operations with the console

unit. These classes are declared in the header file *iostream*. This file should be included in all the programs that communicate with the console unit.

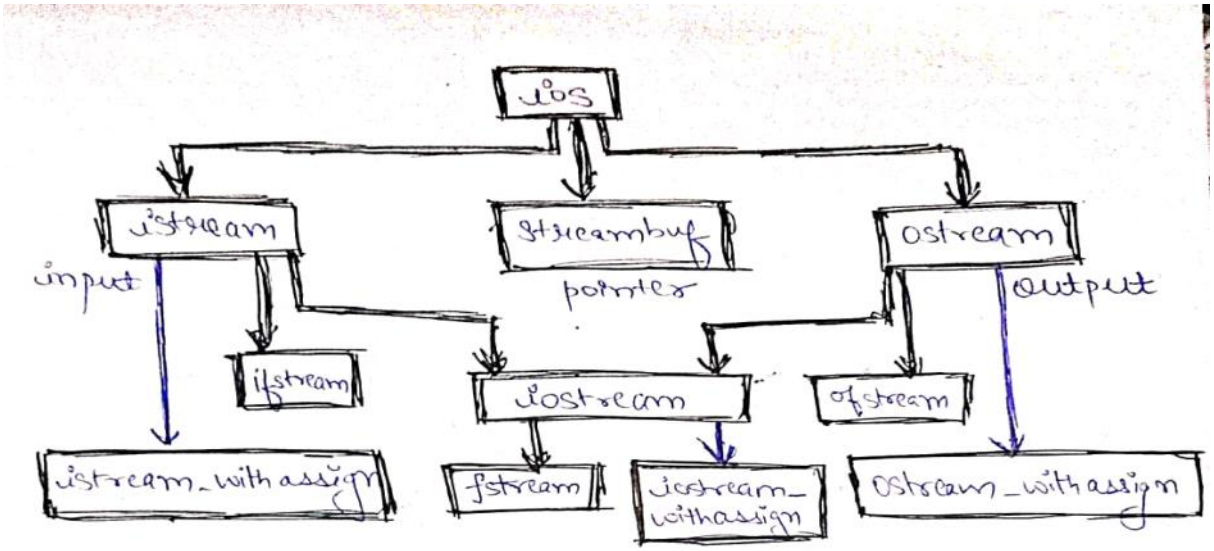


Figure) Stream Classes for console I/O operations

ios is the base class for ***istream*** (input stream) and ***ostream*** (output stream) which are, in turn, base classes for ***iostream*** (input/output stream).

The class ios is declared as the virtual base class so that only one copy of its members are inherited by the iostream. It provides the basic support for formatted and unformatted I/O operations.

The class ***istream*** provides the facilities for formatted and unformatted input while the class ***ostream*** provides the facilities for formatted output(through inheritance).

The class ***iostream*** provides the facilities for handling both input and output streams.

Three classes, namely, ***istream_withassign***, ***ostream_withassign*** and ***iostream_withassign*** add assignment operators to these classes.

Table 10.1 *Stream classes for console operations*

Class name	Contents
ios (General input/output stream class)	• Contains basic facilities that are used by all other input and output classes • Also contains a pointer to a buffer object (**streambuf** object) • Declares constants and functions that are necessary for handling formatted input and output operations
istream (input stream)	• Inherits the properties of **ios** • Declares input functions such as **get(), getline()** and **read()** • Contains overloaded extraction operator >>
ostream (output stream)	• Inherits the properties of ios • Declares output functions **put()** and **write()** • Contains overloaded insertion operator <<
iostream (input/output stream)	• Inherits the properties of **ios istream** and **ostream** through multiple inheritance and thus contains all the input and output functions
streambuf	• Provides an interface to physical devices through buffers • Acts as a base for **filebuf** class used ios files

- **Overloaded >> and << Operators**
  >> is overloaded in istream and << is overloaded in ostream class.
  The operator >> reads the data character by character and assigns it to the indicated location. The reading for a variable will be terminated at the encounter of a white space or a character that does not match the destination type.

```
int code;
cin >> code;
```

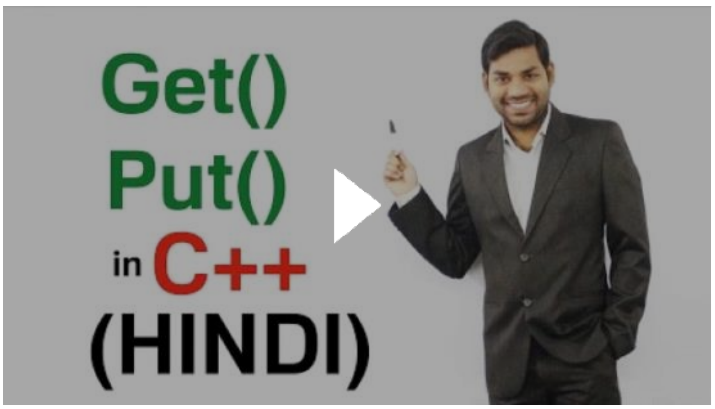
Suppose the following data is given as input:

```
4258D
```

The operator will read the characters upto 8 and the value 4258 is assigned to **code.** The character D remains in the input stream and will be input to the next **cin** statement. The general form for displaying data on the screen is:

- **put() and get() Functions**

Get() and Put() In C++ (HINDI)



The classes ***istream*** and ***ostream*** define two member functions ***get()*** and ***put()*** respectively to handle the single character input/output operations.

There are two of ***get( )*** functions. We can use both get(char *) and get(void) prototypes to fetch a character including the blank space, tab and the newline character. The get(char *) version assigns the input character to its argument and the get(void) version returns the input character.

Since these functions are members of the input/output stream classes. we must invoke them using an appropriate object.

Example:

```
char c;
cin.get(c);             // get a character from keyboard
                        // and assign it to c
while(c != '\n')
{
       cout << c;       // display the character on screen
       cin.get(c);      // get another character
}
```

This code reads and displays a line of text (terminated by a newline character).
Remember, the operator >> can also be used to read a character but it will skip the white spaces and newline character.

The above while loop will not work properly if the statement *cin c;* is used in place of *cin.get(c);*

The *get(void)* version is used as follows:

char c;
c = cin.get(); // cin.get(c); replaced

The value returned by the function *get()* is assignet to the variable c.

The function ***put()*** can be used to output a line of text, character by character.

For Example)
cout.put('x'); displays the character x while *cout.put(ch)* displays the value of variable *ch.*
The variable *ch* must contain a character value.

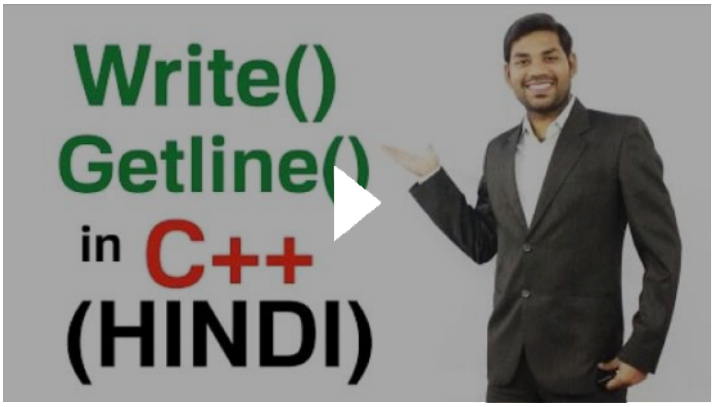
We can also use a number as an argument to the function *put().* For example) *cout.put(68);* displays the character 'D' as per ASCII code.

The following segment of a program reads a line of text from the keyboard and displays it on the screen.

```
char c;
cin.get(c);             // read a character

while(c != '\n')
{
       cout.put(c);     // display the character on screen
       cin.get(c);
}
```

- **getline() and write() Functions**

Getline() and Write() in C++ (HINDI)



We can read and display a line of text more efficiently using the line-oriented input/output functions *getline()* and *write()*.

The ***getline()*** function reads a whole line of text that ends with a newline character (transmitted by the RETURN key). This function can be invoked by using the object *cin* as ***cin.getline(line, size);***

This function call invokes the function *getline()* which reads character input into the variable line.
The reading is terminated as soon as either the newline character '*n*' is encountered or *size* characters are read whichever occurs first.
The newline character is read but not saved. Instead, it is replaced by the null character.

For example, consider the code :

char name[20] ;
cin.getline(name, 20);

Assume that we have given the following input through the keyboard

Bjarne Stroustrup C++ <RETURN>

This input will be read and assigned 'Bjarne Stroustrup C' only to the character array *name* as at max 19 characters can be read (size given as 20 and 1 byte is taken by '\0' null character). Remember, the two blank spaces contained in the string are also taken into account.

<u>*Note*</u>: We can also read strings using the >> operator as *cin >> name;* But *cin* can read strings that do not contain white spaces. This means that *cin* can read just one word and not a series of words or a line(with white spaces). Thus it will store only '*Bjarne*' to the *name*. After reading the string, *cin* automatically adds the terminating null character to the character array.

<u>*Note*</u>: Since *cin* terminates on new line character and does not read new line character. So if *cin* statement is followed by *cin.getline()* statement, cin.getline() will take that new line character(which is waiting in queue buffer as it is not read by *cin*) only as input and insert empty string in the variable.

Example)
```
char arr1[20] , arr2[20];
cin >> arr1; // User inputs 'Delhi <RETURN>' will store 'Delhi' to arr1.
cin.getline(arr2,20);
/* Then this statement will not wait for user input as it will take
```

<RETURN> from input buffer and store empty string in arr2. */

The ***write()*** function displays an entire line and has the following form:

```
char* line = "Bjarne Stroustrup C++";
cout.write (line, size);
```

The first argument *line* the name of the string to displayed and the second argument *size* indicates the number of characters to display.

Note: It does not stop displaying the characters automatically when the null character is encountered. If the *size* is greater than the length of *line*, then it displays beyond the bounds of line.
Although if length of *line* is greater than the *size* passed, then only first (size - 1) characters of *line* will be displayed (1 less because of null '\0' character).

Note: It is possible to ***concatenate*** two strings (display them combined) using the *write()* function in a single statement :

```
cout.write(str1, size1).write(str2, size2);
```

It is possible because write function of cout object itself returns a cout object.
This statement is equivalent to

```
cout.write(str1, size1);
cout.write(str2, size2);
```

- ***ios class functions and flags***
  The ios class contains a large number of member functions that would help us to format the output in a number of ways. The most important ones among them are *width(), precision(), fill(), setf() and unsetf()*;

Table 10.2 ios *format functions*

Function	*Task*
Width ()	To specify the required field size for displaying an output value
precision ()	To specify the number of digits to be displayed after the decimal point of a float value
fill()	To specify a character that is used to fill the unused portion of a field
setf()	To specify format flags that can control the form of output display (such as left-justification and right-justification)
unsetf()	To clear the flags specified

1. width()

Defining Field Width: width()

We can use the **width()** function to define the width of a field necessary for the output of an item. Since, it is a member function, we have to use an object to invoke it, as shown below:

```
cout.width(w);
```

where w is the field width (number of columns). The output will be printed in a field of w characters wide at the right end of the field. The **width()** function can specify the field width for only one item (the item that follows immediately). After printing one item (as per the specifications) it will revert back to the default. For example, the statements

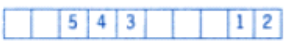
```
cout.width(5);
cout << 543  <<  12 << "\n";
```

will produce the following output:

The value 543 is printed right-justified in the first five columns. The specification width(5) does not retain the setting for printing the number 12. This can be improved as follows:

```
cout.width(5);
cout << 543;
cout.width(5);
cout << 12  << "\n";
```

This produces the following output:



Remember that the field width should be specified for each item separately. C++ never truncates the values and therefore, if the specified field width is smaller than the size of the value to be printed, C++ expands the field to fit the value. Program 10.4 demonstrates how the function **width()** works.

```
int main()
{
  int items[4] = {10,8,12,15};
  int cost[4] = {75,100,60,99};

  cout.width(5);
  cout <<  "ITEMS";
  cout.width(8);
  cout << "COST";

  cout.width(15);
  cout << "TOTAL VALUE"  << "\n";

  int sum = 0;

  for(int i=0; i<4; i++)
  {
        cout.width(5);
        cout << items[i];

        cout.width(8);
        cout << cost[i];

        int value = items[i] * cost[i];
        cout.width(15);
        cout << value << "\n";
        sum = sum + value;
  }
   cout << "\n Grand Total = ";

  cout.width(2);
  cout << sum << "\n";

  return 0;
}
```

The output of Program 10.4 would be:

```
     ITEMS          COST  TOTAL VALUE

        10            75          750
         8           100          800
        12            60          720
        15            99         1485

     Grand Total = 3755
```

note

A field of width two has been used for printing the value of sum and the result is not truncated. A good gesture of C++ !

2. precision()

Setting Precision: precision()

By default, the floating numbers are printed with six digits after the decimal point. However, we can specify the number of digits to be displayed after the decimal point while printing the floating-point numbers. This can be done by using the **precision()** member function as follows:

```
cout.precision(d);
```

where d is the number of digits to the right of the decimal point. For example, the statements

```
cout.precision(3);
cout << sqrt(2) << "\n";
cout << 3.14159 << "\n";
cout << 2.50032 << "\n";
```

will produce the following output:

```
1.141   (truncated)
3.142   (rounded to the nearest cent)
2.5     (no trailing zeros)
```

Not that, unlike the function **width()**, **precision()** retains the setting in effect until it is reset. That is why we have declared only one statement for the precision setting which is used by all the three outputs.

We can set different values to different precision as follows:

```
cout.precision(3);

cout << sqrt(2)  << "\n";
cout.precision(5);              // Reset the precision
cout << 3.14159  << "\n";
```

We can also combine the field specification with the precision setting. Example:

```
cout.precision(2);
cout.width(5);
cout << 1.2345;
```

The first two statements instruct: "print two digits after the decimal point in a field of five character width". Thus, the output will be:

	1		2	3

Program 10.5 shows how the functions **width()** and **precision()** are jointly used to control the output format.

PRECISION SETTING WITH precision()

```
#include <iostream>
#include <cmath>

using namespace std;

int main()
{
        cout << "Precision set to 3 digits \n\n";
        cout.precision(3);

        cout.width(10);
        cout << "VALUE";
        cout.width(15);
        cout << "SQRT OF VALUE" << "\n";

        for(int n=1; n<=5; n++)
        {
                cout.width(8);
                cout << n;
                cout.width(13);
                cout << sqrt(n)  << "\n";
        }
```

```
    cout << "\n Precision set to 5 digits \n\n";
    cout.precision(5);              // precision parameter changed
    cout << " sqrt(10) = " << sqrt(10) << "\n\n";

    cout.precision(0);              // precision set to default
    cout << " sqrt(10) = " << sqrt(10) << " (default setting)\n";

    return 0;
}
```

PROGRAM 10.5

Here is the output of Program 10.5

```
Precision set to 3 digits
       VALUE    SQRT_OF_VALUE
           1                1
           2             1.41
           3             1.73
           4                2
           5             2.24

Precision set to 5 digits

sqrt(10) = 3.1623

sqrt(10) = 3.162278 (default setting)
```

note

Observe the following from the output:

1. The output is rounded to the nearest cent (i.e., 1.6666 will be 1.67 for two digit precision but 1.3333 will be 1.33).
2. Trailing zeros are truncated.
3. Precision setting stays in effect until it is reset.
4. Default precision is 6 digits.

3. fill()

Filling and Padding: fill()

We have been printing the values using much larger field widths than required by the values. The unused positions of the field are filled with white spaces, by default. However, we can use the fill() function to fill the unused positions by any desired character. It is used in the following form:

```
cout.fill (ch);
```

Where *ch* represents the character which is used for filling the unused positions. Example:

```
cout.fill('*');
cout.width(10);
cout << 5250  << "\n";
```

The output would be:

*	*	*	*	*	*	5	2	5	0

Financial institutions and banks use this kind of padding while printing cheques so that no one can change the amount easily. Like **precision()**, **fill()** stays in effect till we change it. See Program 10.6 and its output.

PADDING WITH fill()

```
#include <iostream>

using namespace std;

int main( )
{       cout.fill('<');

        cout.precision(3);
        for(int n=1; n<=6; n++)
        {
            cout.width(5);
            cout << n;
            cout.width(10);
            cout << 1.0 / float(n) << "\n";
            if (n == 3)
               cout.fill ('>');
        }
        cout << "\nPadding changed \n\n";
        cout.fill ('#');     // fill( ) reset
        cout.width (15);
        cout << 12.345678    << "\n";

        return 0;
}
```

PROGRAM 10.6

The output of Program 10.6 would be:

```
<<<<1<<<<<<<<<1
<<<<2<<<<<<<0.5
<<<<3<<<<<0.333
>>>>4>>>>>>0.25
>>>>5>>>>>>>0.2
>>>>6>>>>>0.167

Padding changed

########12.346
```

4. setf()

Formatting Flags, Bit-fields and setf()

We have seen that when the function **width**() is used, the value (whether text or number) is printed right-justified in the field width created. But, it is a usual practice to print the text left-justified. How do we get a value printed left-justified? Or, how do we get a floating-point number printed in the scientific notation?

The **setf**(), a member function of the **ios** class, can provide answers to these and many other formatting questions. The **setf**() (*setf* stands for set flags) function can be used as follows:

```
cout.setf(arg1,arg2)
```

The *arg1* is one of the formatting *flags* defined in the class **ios**. The formatting flag specifies the format action required for the output. Another **ios** constant, *arg2*, known as bit *field* specifies the group to which the formatting flag belongs.

Table 10.4 shows the bit fields, flags and their format actions. There are three bit fields and each has a group of format flags which are mutually exclusive. Examples:

```
cout.setf(ios::left, ios::adjustfield);
cout.setf(ios::scientific, ios::floatfield);
```

Note that the first argument should be one of the group members of the second argument.

Consider the following segment of code:

```
cout.fill('*');
cout.setf(ios::left, ios::adjustfield);
cout.width(15);
cout << "TABLE 1" <<  "\n";
```

Table 10.4 *Flags and bit fields for setf() function*

Format required	*Flag (arg1)*	*Bit-field (arg2)*
Left-justified output	ios :: left	ios :: adjustfield
Right-justified output	ios :: right	ios :: adjustfield
Padding after sign or base Indicator (like +##20)	ios :: internal	ios :: adjustfield
Scientific notation	ios :: scientific	ios :: floatfield
Fixed point notation	ios :: fixed	ios :: floatfield
Decimal base	ios :: dec	ios :: basefield
Octal base	ios :: oct	ios :: basefield
Hexadecimal base	ios :: hex	ios :: basefield

This will produce the following output:

T	A	B	L	E		1	*	*	*	*	*	*	*	*

The statements

```
cout.fill ('*');
cout.precision(3);
cout.setf(ios::internal, ios::adjustfield);
cout.setf(ios::scientific, ios::floatfield);
cout.width(15);

cout << -12.34567  <<'"\n";
```

will produce the following output:

-	*	*	*	*	*	1	.	2	3	5	e	+	0	1

note

The sign is left-justified and the value is right left- justified. The space between them is padded with stars. The value is printed accurate to three decimal places in the scientific notation.

Displaying Trailing Zeros and Plus Sign

If we print the numbers 10.75, 25.00 and 15.50 using a field width of, say, eight positions, with two digits precision, then the output will be as follows:

			1	0	.	7	5
						2	5
				1	5	.	5

Note that the trailing zeros in the second and third items have been truncated.

Certain situations, such as a list of prices of items or the salary statement of employees, require trailing zeros to be shown. The above output would look better if they are printed as follows:

```
10.75
25.00
15.50
```

The **setf()** can be used with the flag **ios::showpoint** as a single argument to achieve this form of output. For example,

```
cout.setf(ios::showpoint);        // display trailing zeros
```

would cause cout to display trailing zeros and trailing decimal point. Under default precision, the value 3.25 will be displayed as 3.250000. Remember, the default precision assumes a precision of six digits.

Similarly, a plus sign can be printed before a positive number using the following statement:

```
cout.setf(ios::showpos);        // show +sign
```

For example, the statements

```
cout.setf(ios::showpoint);
cout.setf(ios::showpos);
cout.precision(3);
cout.setf(ios::fixed, ios::floatfield);
cout.setf(ios::internal, ios::adjustfield);
cout.width(10);
cout << 275.5 << "\n";
```

will produce the following output:

+			2	7	5	.	5	0	0

The flags such as **showpoint** and **showpos** do not have any bit fields and therefore are used as single arguments in **setf()**. This is possible because the **setf()** has been declared as an overloaded function in the class **ios**. Table 10.5 lists the flags that do not possess a named bit field. These flags are not mutually exclusive and therefore can be set or cleared independently.

Table 10.5 *Flags that do not have bit fields*

Flag	*Meaning*
ios :: showbase	Use base indicator on output
ios :: showpos	Print + before positive numbers
ios :: showpoint	Show trailing decimal point and zeroes
ios :: uppercase	Use uppercase letters for hex output
ios :: skipus	Skip white space on input
ios :: unitbuf	Flush all streams after insertion
ios :: stdio	Flush **stdout** and **stderr** after insertion

Program 10.7 demonstrates the setting of various formatting flags using the overloaded **setf()** function.

FORMATTING WITH FLAGS IN setf()

```
#include <iostream>
#include <cmath>

using namespace std;

int main()
{
        cout.fill('*');
        cout.setf(ios::left, ios::adjustfield);
        cout.width(10);
        cout << "VALUE";

        cout.setf(ios::right, ios::adjustfield);
        cout.width(15);
        cout << "SQRT OF VALUE" << "\n";

        cout.fill('.');
        cout.precision(4);
        cout.setf(ios::showpoint);
        cout.setf(ios::showpos);
        cout.setf(ios::fixed, ios::floatfield);

        for(int n=1; n<=10; n++)
        {
                cout.setf(ios::internal, ios::adjustfield);
                cout.width(5);
                cout << n;

                cout.setf(ios::right, ios::adjustfield);
                cout.width(20);
                cout << sqrt(n)  << "\n";
        }

        // floatfield changed
        cout.setf(ios::scientific, ios::floatfield);
        cout << "\nSQRT(100) = " << sqrt(100) << "\n";

        return 0;
}
```

PROGRAM 10.7

The output of Program 10.7 would be:

```
VALUE*********SQRT OF VALUE
+...1..............+1.0000
+...2..............+1.4142
+...3..............+1.7321
+...4..............+2.0000
+...5..............+2.2361
+...6..............+2.4495
+...7..............+2.6458
+...8..............+2.8284
+...9..............+3.0000
+..10..............+3.1623

SQRT(100) = +1.0000e+001
```

note

1. The flags set by **setf()** remain effective until they are reset or unset.
2. A format flag can be reset any number of times in a program.
3. We can apply more than one format controls jointly on an output value.
4. The setf() sets the specified flags and leaves others unchanged.

- ***Manipulators***

  Manipulators are special functions that are included in the I/O statements to alter the format parameters of a stream. Some important manipulators are *setw(), setprecision(), setfill(), setiosflags() and resetiosflags()*. To access these manipulators, the file ***iomanip*** should included in the program. They provide same features as that of *ios* member functions and flags but are more convenient to use.

Table 10.3 *Manipulators*

Manipulators	Equivalent ios function
setw()	width()
setprecision()	precision()
setfill()	fill()
setiosflags()	setf()
resetiosflags()	unsetf()

Table 10.6 *Manipulators and their meanings*

Manipulator	Meaning	Equivalent
setw (int *w*)	Set the field width to *w*.	width( )
setprecision(int *d*)	Set the floating point precision to *d*.	precision( )
setfill(int *c*)	Set the fill character to *c*.	fill( )
setiosflags(long *f*)	Set the format flag *f*.	setf( )
resetiosflags(long *f*)	Clear the flag specified by *f*.	unsetf( )
endl	Insert new line and flush stream.	"\n"

Some examples of manipulators are given below:

```
cout << setw(10) << 12345;
```

This statement prints the value 12345 right-justified in a field width of 10 characters. The output can be made left-justified by modifying the statement as follows:

```
cout << setw(10) << setiosflags(ios::left) << 12345;
```

One statement can be used to format output for two or more values. For example, the statement

```
cout   << setw(5)  << setprecision(2) << 1.2345
       << setw(10) << setprecision(4) << sqrt(2)
       << setw(15) << setiosflags(ios::scientific) << sqrt(3);
       << endl;
```

will print all the three values in one line with the field sizes of 5, 10, and 15 respectively. Note that each output is controlled by different sets of format specifications.

We can jointly use the manipulators and the **ios** functions in a program. The following segment of code is valid:

```
cout.setf(ios::showpoint);
cout.setf(ios::showpos);
cout << setprecision(4);
cout << setiosflags(ios::scientific);
cout << setw(10) << 123.45678;
```

Note: There is a major difference in the way the manipulators are implemented as compared to the *ios* member functions. The ios member function return the previous format state which can be used later, if necessary. But the manipulator does not return the previous format state. In case. we need to save the old format states. we must use the ios member functions rather than the manipulators.

Types of Manipulators

There are various types of manipulators:

1. **Manipulators without arguments**: The most important manipulators defined by the **Iostream library** are provided below.
    - **endl**: It is defined in ostream. It is used to enter a new line and after entering a new line it

flushes (i.e. it forces all the output written on the screen or in the file) the output stream.
- **ws**: It is defined in istream and is used to ignore the whitespaces in the string sequence.
- **ends**: It is also defined in ostream and it inserts a null character into the output stream. It typically works with std::ostrstream, when the associated output buffer needs to be null-terminated to be processed as a C string.
- **flush**: It is also defined in ostream and it flushes the output stream, i.e. it forces all the output written on the screen or in the file. Without flush, the output would be the same, but may not appear in real-time.

```
#include <iostream>
#include <istream>
#include <sstream>
#include <string>

using namespace std;

int main()
{
    istringstream str("           Programmer");
    string line;
    // Ignore all the whitespace in string
    // str before the first word.
    getline(str >> std::ws, line);

    // you can also write str>>ws
    // After printing the output it will automatically
    // write a new line in the output stream.
    cout << line << endl;

    // without flush, the output will be the same.
    cout << "only a test" << flush;

    // Use of ends Manipulator
    cout << "\na";

    // NULL character will be added in the Output
    cout << "b" << ends;
    cout << "c" << endl;

    return 0;
}
```

Output:

```
Programmer
only a test
abc
```

2. **Manipulators with Arguments:** Some of the manipulators are used with the argument like setw (20), setfill ('*'), and many more. These all are defined in the header file. If we want to use these manipulators then we must include this header file in our program.
   For Example, you can use following manipulators to set minimum width and fill the empty space with any character you want: std::cout << std::setw (6) << std::setfill ('*');

- **Some important manipulators in <*iomanip*> are:**
  1. **setw (val):** It is used to set the field width in output operations.
  2. **setfill (c):** It is used to fill the character 'c' on output stream.
  3. **setprecision (val):** It sets val as the new value for the precision of floating-point values.
  4. **setbase(val):** It is used to set the numeric base value for numeric values.
  5. **setiosflags(flag):** It is used to set the format flags specified by parameter mask.
  6. **resetiosflags(m):** It is used to reset the format flags specified by parameter mask.

- **Some important manipulators in <ios> are:**
  1. **showpos:** It forces to show a positive sign on positive numbers.
  2. **noshowpos:** It forces not to write a positive sign on positive numbers.
  3. **showbase:** It indicates the numeric base of numeric values.
  4. **uppercase:** It forces uppercase letters for numeric values.

5. **nouppercase:** It forces lowercase letters for numeric values.
6. **fixed:** It uses decimal notation for floating-point values.
7. **scientific:** It uses scientific floating-point notation.
8. **hex:** Read and write hexadecimal values for integers and it works same as the setbase(16).
9. **dec:** Read and write decimal values for integers i.e. setbase(10).
10. **oct:** Read and write octal values for integers i.e. setbase(10).
11. **left:** It adjusts output to the left.
12. **right:** It adjusts output to the right.

```
#include <iomanip>
#include <iostream>
using namespace std;

int main()
{
    double A = 100;
    double B = 2001.5251;
    double C = 201455.2646;

    // We can use setbase(16) here instead of hex

    // formatting
    cout << hex << left << showbase << nouppercase;

    // actual printed part
    cout << (long long)A << endl;

    // We can use dec here instead of setbase(10)

    // formatting
    cout << setbase(10) << right << setw(15)
         << setfill('_') << showpos
         << fixed << setprecision(2);

    // actual printed part
    cout << B << endl;

    // formatting
    cout << scientific << uppercase
         << noshowpos << setprecision(9);

    // actual printed part
    cout << C << endl;
}
```

Output:

```
0x64
_______+2001.53
2.014552646E+05
```

- ***User-defined output functions***

  In addition to these functions supported by the C++ library, we can create our own manipulator functions to provide any special output formats.

  In C++, user can define and use manipulator similar to built in manipulators as per the users' used and desire to control the format of input and output. Similar to the predefined built manipulators use can define non parameterized as well as parameterized manipulators.

  The syntax for designing manipulator is as follows:

  ostream& manipulator_name(ostream& output, args..)
  {
  //body of user defined manipulators

```
    return output;
}
```

The *manipulator_name* is the name of the user defined manipulator, ostream & ouput is the manipulator called and stream cascading object and args is the number of arguments for parameterized manipulators.

Example)

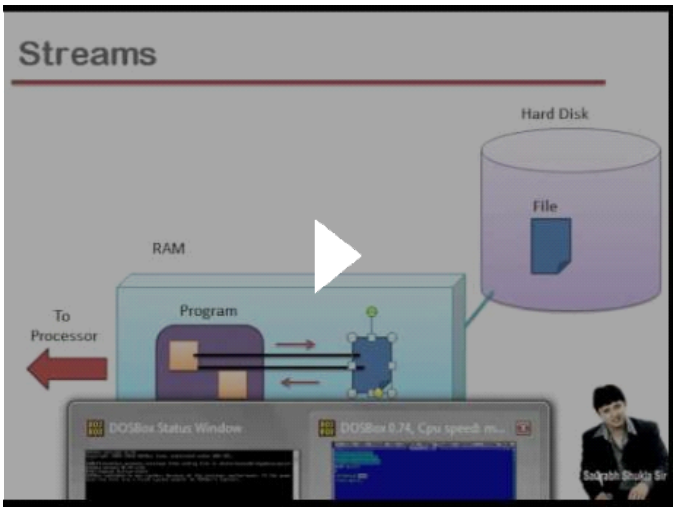
```
#include < iostream.h>
#include < iomanip.h>

ostream&curr(ostream&ostrObj)
{
    cout << fixed << setprecision(2);
    cout << "Rs.";
    returnostrObj;
}

void main()
{
    floatamt = 10.5478;
    cout << curr << amt;
}
```

Data File Handling in C++

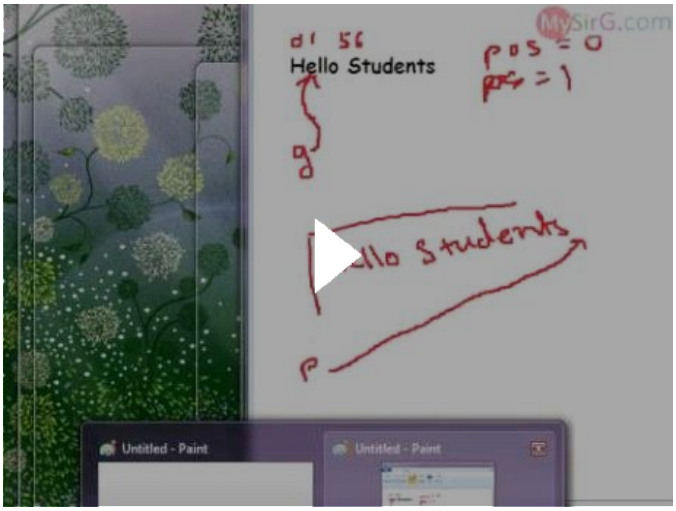
Lecture 21 File Handling in C++ Part 1 Hindi

Lecture 21 File Handling in C++ Part 2 Hindi

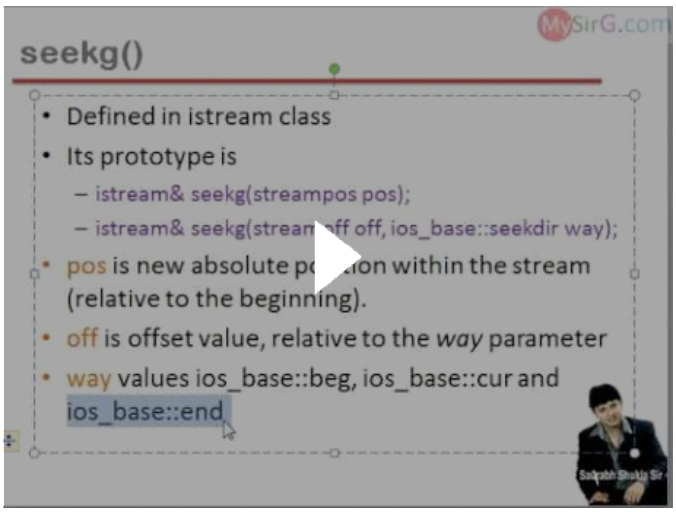
Lecture 21 File Handling in C++ Part 3 Hindi



tellg and tellp in C++ language Hindi



seekg and seekp functions in C++ Hindi



A **file** is a collection of related data stored in a particular area on the disk. Programs can be designed to perform the read and write operations on these files.

A program typically involves either or of the following kinds of data communication:

1. Data transfer between the console unit and the program.
2. Data transfer between the program and a disk file.

I/O system of C++ handles file operations which are very similar to the console input and output operations i.e. by using *streams* as an interface.

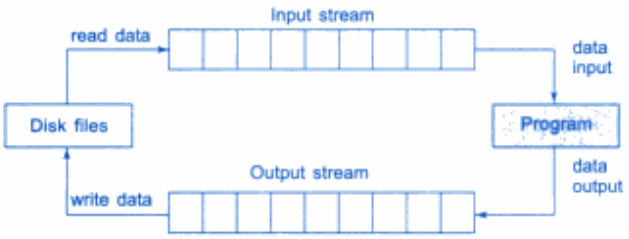


Fig. 11.2 ⇔ *File input and output streams*

The input operation involves the creation of an *input stream* and linking it with the program and the input file. Similarly, the output operation involves establishing an *output stream* with the links with the program and the output file.

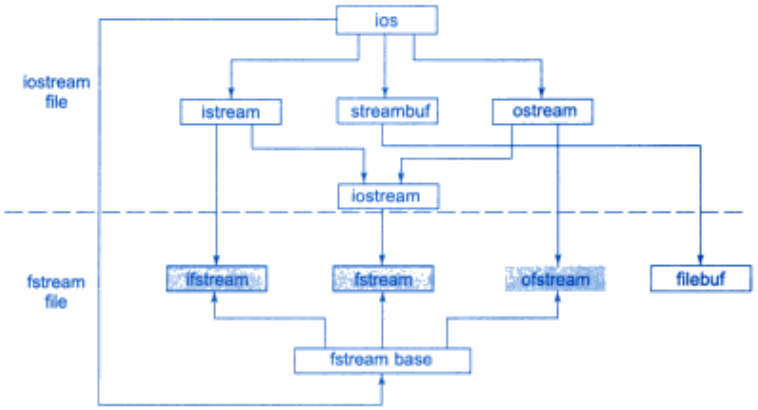


Fig. 11.3 ⇔ *Stream classes for file operations (contained in fstream file)*

Opening & Closing Files

To use a disk file, following things need to be decided:

1. ***Filename*** is a string of characters that make up a valid name of a disk file for the operating system. It contain two parts: a primary name and an optional period (.) with extension.
   Eg) student, teacher.txt, test.doc, input.data , etc.

2. ***Filestream*** must be created and linked to the *filename*. It can be defined using the classes *ifstream*, *ofstream* or *fstream*.

Table 11.1 *Details of file stream classes*

Class	Contents
`filebuf`	Its purpose is to set the file buffers to read and write. Contains **Openprot** constant used in the **open()** of file stream classes. Also contain **close()** and **open()** as members.
`fstreambase`	Provides operations common to the file streams. Serves as a base for **fstream, ifstream** and **ofstream** class. Contains **open()** and **close()** functions.
`ifstream`	Provides input operations. Contains **open()** with default input mode. Inherits the functions **get(), getline(), read(), seekg()** and **tellg()** functions from **istream**.
`Ofstream`	Provides output operations. Contains **open()** with default output mode. Inherits **put(), seekp(), tellp()**, and **write()**, functions from **ostream**.
`fstream`	Provides support for simultaneous input and output operations. Contains **open()** with default input mode. Inherits all the functions from **istream** and **ostream** classes through **iostream**.

3. Class to be used depends on the **purpose**, that is whether we need to ***read*** data from file or ***write*** data into file.

4. **File Opening Method**: File can be opened in two ways:

   a. Using the ***constructor*** function of the class : useful when only one file need to be read/write in the stream

      It involves the following steps:
      i. Create a file stream object to manage the input or output stream using the appropriate class (istream/ostream/fstream)
      ii. Initialize the object with the desired filename as parameter to the constructor.

      Syntax) *streamClass streamObject("filename");*
      For eg) *ofstream fout("results.dat"); // writing data into file*
      ifstream fin("results.dat"); // reading data from file

   b. Using the member function ***open()*** of the class : useful when we want to manage multiple files using single stream
      Syntax) *streamClass streamObject;*
      streamObject.open("filename");
      For Eg) *ofstream fout;*
      fout.open("results.dat");

      Note: We need to close the file before opening another file if using the same streamObject. This is necessary because a **stream can be connected to only one file at a time**.
      We can close a file i.e. disconnect the file from the stream (streamObject) using *streamObject.close()* function. It is a good practice to close a file everytime it is used in a program as it makes the stream free to be connected to another file.

5. **File Mode Parameter**: Both methods of file opening can take another(second) optional parameter, the second one for specifying the file mode.

File mode parameter specifies the purpose for which the file is opened.
If no second parameter is passed, then the default values are:

=> ios::in for ifstream i.e. open for read only

=> ios::out for ofstream i.e. open for write only (create file if don't exist but erase content if already exist)

Table 11.2 *File mode parameters*

Parameter	Meaning
ios :: app	Append to end-of-file
ios :: ate	Go to end-of-file on opening
ios :: binary	Binary file
ios :: in	Open file for reading only
ios :: nocreate	Open fails if the file does not exist
ios :: noreplace	Open fails if the file already exists
ios :: out	Open file for writing only
ios :: trunc	Delete the contents of the file if it exists

note

1. Opening a file in **ios::out** mode also opens it in the **ios::trunc** mode by default.
2. Both **ios::app** and **ios::ate** take us to the end of the file when it is opened. The difference between the two parameters is that the **ios::app** allows us to add data to the end of the file only, while **ios::ate** mode permits us to add data or to modify the existing data anywhere in the file. In both the cases, a file is created by the specified name, if it does not exist.
3. The parameter **ios::app** can be used only with the files capable of output.
4. Creating a stream using **ifstream** implies input and creating a stream using **ofstream** implies output. So in these cases it is not necessary to provide the mode parameters.
5. The **fstream** class does not provide a mode by default and therefore, we must provide the mode explicitly when using an object of **fstream** class.
6. The *mode* can combine two or more parameters using the bitwise OR operator (symbol |) as shown below:

```
fout.open("data", ios::app | ios:: nocreate)
```

This opens the file in the append mode but fails to open the file if it does not exist.

Detecting End of File

Detection of the end-of-file condition is necessary for preventing any further attempt to read data from the file.

Methods:

1. ***if(fin)*** cout << "End of File not reached"; else cout << "End of File reached";
   An ifstream object such as fin returns a value of 0 if any error occurs in the file operation including the end-of-file condition.
2. ***if(!fin.eof())*** cout << "End of File not reached"; else cout << "End of File reached";
   eof() is a member function of *ios* class. It returns a non zero value if the end-of-file (EOF) condition is encountered, and a zero, otherwise.

File Pointers

Each file has two associaed pointers known as *file pointers*. These are input pointer (or *get* pointer) and output pointer (or *put* pointer). These pointers are used to move through the files while reading and writing.

The input pointer is used for reading thr contents of a given line location and the output pointer for writing to the given file location. Reading/writing operations automatically advances the appropriate pointers to the new location.

Default Actions

When we a file in read-only mode, the input pointer is automatically set at the beginning so that we can read the file from the start. Similarly, when we open a file in write-only mode, the existing contents are deleted and output pointer is at the beginning. This enables us to write to the file from the start. In case. we want to open an existing file to add more data, the file is opened in '*append*' mode. This moves the output pointer to the end of the file (i.e. the end of the existing contents)

Functions for Manipulation of File Pointers

All the actions on the file pointers as shown in Fig. 11.7 take place automatically by default. How do we then move a file pointer to any other desired position inside the file? This is possible only if we can take control of the movement of the file pointers ourselves. The file stream classes support the following functions to manage such situations:

- **seekg()** Moves get pointer (input) to a specified location.
- **seekp()** Moves put pointer(output) to a specified location.
- **tellg()** Gives the current position of the get pointer.
- **tellp()** Gives the current position of the put pointer.

For example, the statement

```
infile.seekg(10);
```

moves the file pointer to the byte number 10. Remember, the bytes in a file are numbered beginning from zero. Therefore, the pointer will be pointing to the 11th byte in the file.

Consider the following statements:

```
ofstream  fileout;
fileout.open("hello", ios::app);
int p = fileout.tellp();
```

On execution of these statements, the output pointer is moved to the end of the file "hello" and the value of **p** will represent the number of bytes in the file.

Specifying the offset

We have just now seen how to move a file pointer to a desired location using the 'seek' functions. The argument to these functions represents the absolute position in the file. This is shown in Fig. 11.8.

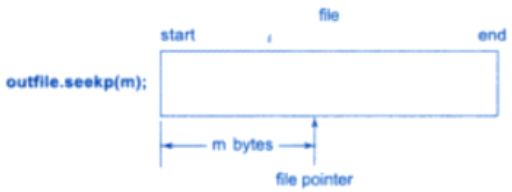


Fig. 11.8 ⇔ *Action of single argument seek function*

'Seek' functions **seekg()** and **seekp()** can also be used with two arguments as follows:

```
seekg (offset, refposition);
seekp (offset, refposition);
```

The parameter *offset* represents the number of bytes the file pointer is to be moved from the location specified by the parameter *refposition*. The *refposition* takes one of the following three constants defined in the **ios** class:

- **ios::beg** start of the file
- **ios::cur** current position of the pointer
- **ios::end** End of the file

The **seekg()** function moves the associated file's 'get' pointer while the **seekp()** function moves the associated file's 'put' pointer. Table 11.3 lists some sample pointer offset calls and their actions. **fout** is an **ofstream** object.

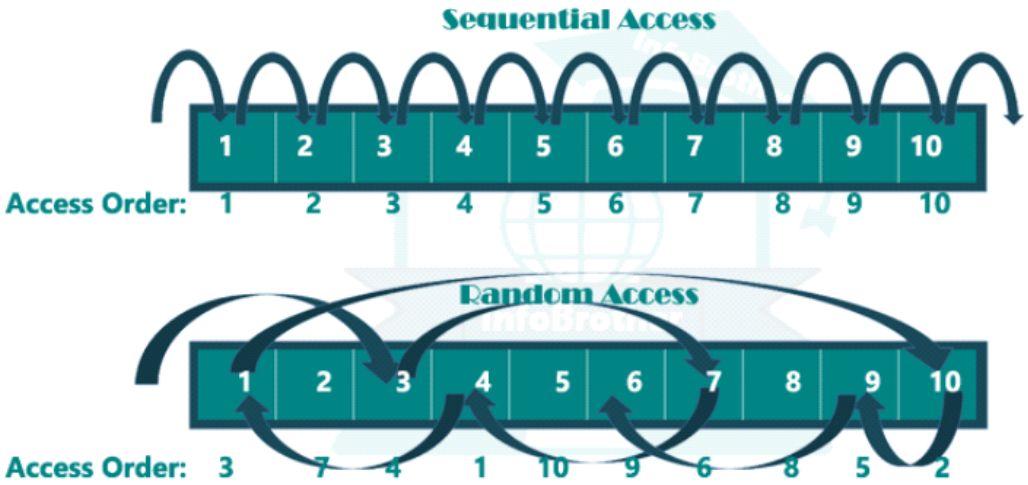
Table 11.3 *Pointer offset calls*

Seek call	*Action*
fout.seekg(o, ios::beg);	Go to start
fout.seekg(o, ios::cur);	Stay at the current position
fout.seekg(o, ios::end);	Go to the end of file
Fout.seekg(m,ios::beg);	Move to (m + 1)th byte in the file
fout.seekg(m,ios::cur);	Go forward by m byte form the current position
fout.seekg(-m,ios::cur);	Go backward by m bytes from the current position
fout.seekg(-m,ios::end);	Go backward by m bytes form the end

Sequential I/O Operations

The file stream classes support a number of member functions for performing the input and output operations on files. The functions ***get() and put()*** are capable of handling a single character at a time. The function ***getline()*** lets you handle multiple characters at a time. Another pair of functions i.e., ***read() and write()*** are capable of reading and writing blocks of binary data.

Please Refer: https://codescracker.com/cpp/cpp-sequential-io-with-files.htm



Sequential files do not lend themselves to quick access. It is not feasible, in many situations, to look up individual records in a data file with sequential access.

Random Access I/O Operations

Unlike Sequential files, we can access Random Access files in any order we want. Random-file access

sometimes takes more Programming but rewards our effort with a more flexible file-access method, power and speed of disk access.
Random file access enables us to read or write any data in our disk file without having to read or write every piece of data before it. We can ***read/write data, search data, modify data, delete data***, in a random- access file. We can open and close Random access file similarly like Sequential files with same opening mode, but we need a few new functions to access files randomly. Functions in C++ which can help move *file pointer* in random manner are : ***seekg(), seekp() , tellg() and tellp()***.

Special function to move File-Pointer within the File:		
Function	**Syntax**	**Explanation**
seekg()	`fileObject.seekg(long_num, origin);`	We can move input pointer to a specified location for reading by using this function. fileObject is the pointer to the file that we want to access. long_num is the number of bytes in the file we want to skip. and origin is a value that tells to compiler, where to begin the skipping of bytes.
seekp()	`fileObject.seekp(long_num, origin);`	We can move Output pointer to a specified location by using this function. it's same like seekg() Function but in Case of writing:
tellg()	`fileObject.tellg();`	This function return the current position of Input pointer. this function don't need any argument.
tellp()	`fileObject.tellp();`	This function return the current position of Output pointer. this function don't need any argument.

Note: *Origin* is the value that tells compiler, where to begin the skipping of bytes specified by *long_num. Origin* can be of 3 types :

Origin	Syntax	Explanation
ios::beg	`fileObject.seekg(0, ios::beg);`	Go to Start: No matter how far into a file we have read, by using this Syntax, the file-pointer will back to the beginning of the file.
ios::cur	`fileObject.seekg(0, ios::cur);`	Stay at Current Position: Using this syntax, the file-pointer will show its current position.
ios::end	`fileObject.seekg(0, ios::end);`	Go to End of the file: using this syntax, the file-pointer will point to end of the file.

Error Handling in File Operations
Sometimes during file operations, errors may also creep in. For example, a file being opened for reading might not exist. Or a file name used for a new file may already exist. Or an attempt could be made to read past the end-of-file. Or such as invalid operation may be performed. There might not be enough space in the disk for storing data.
To check for such errors and to ensure smooth processing, C++ file streams inherit 'stream-state'

members from the ios class that store the information on the status of a file that is being currently used. The current state of the I/O system is held in an integer, in which the following flags are encoded :

Name	Meaning
eofbit	1 when end-of-file is encountered, 0 otherwise.
failbit	1 when a non-fatal I/O error has occurred, 0 otherwise
badbit	1 when a fatal I/O error has occurred, 0 otherwise
goodbit	0 value

C++ Error Handling Functions

There are several error handling functions supported by class ios that help you read and process the status recorded in a file stream.
Following table lists these error handling functions and their meaning :

Function	Meaning
int bad()	Returns a non-zero value if an invalid operation is attempted or any unrecoverable error has occurred. However, if it is zero (false value), it may be possible to recover from any other error reported and continue operations.
int eof()	Returns non-zero (true value) if end-of-file is encountered while reading; otherwise returns zero (false value).
int fail()	Returns non-zero (true) when an input or output operation has failed.
int good()	Returns non-zero (true) if no error has occurred. This means, all the above functions are false. For example, if fin.good() is true, everything is okay with the stream named as fin and we can proceed to perform I/O operations. When it returns zero, no further operations can be carried out.
clear()	Resets the error state so that further operations can be attempted.

The above functions can be summarized as eof() returns true if eofbit is set; bad() returns true if badbit is set. The fail() function returns true if failbit is set; the good() returns true there are no errors, otherwise, they return false.

Command Line Arguments

Lecture 36 Command Line Arguments in C language Hindi

Procedural (or Structural) Programming

- The basic principle of the structured programming approach is to divide a program into functions and modules that perform specific tasks. The use of modules and functions makes the program code cleaner , more understandable and readable.
- This approach is also known as the **top-down approach**. A top-down approach begins with high level design and ends with low level design or development.
- Data is global, and all the functions can access global data i.e. data move openly around the system from function to function. The basic **drawback** of the procedural programming approach is that data is not secured because data is global and can be accessed by any function.
- This approach gives importance to functions rather than data. It focuses on the development of medium to large software applications, for example, C was used for modern operating system development. The programming languages: PASCAL, C and FORTRAN
- It does not model real world problems and its applications really well as there are no OOP features like Inheritance , Data hiding and encapsulation , etc.

Object Oriented Programming

Object Oriented programming (OOP) is a programming paradigm that relies on the concept of classes and objects. It is used to structure a software program into simple, reusable pieces of code blueprints (usually called classes), which are used to create individual instances of objects.

AIM *:* "Object-oriented programming aims to implement real-world entities like inheritance, hiding, polymorphism, etc in programming. The main aim of OOP is to bind together the data and the functions that operate on them so that no other part of the code can access this data except that function ."

Object-oriented programming (OOP) allows the writing of programs with the help of certain classes and real-time objects. This approach is very close to the real-world and its applications because the state and behavior of these classes and objects are almost the same as real-world objects. OOP programming languages: C++ and JAVA.

Note: **Object-based programming** approach is the approach which primarily supports only few OOP concepts like data encapsulation , data hiding, function overloading and objects. This approach don't implement inheritance and dynamic binding. Eg of OBP language : Ada programming language.
Thus Object Oriented Programming = Object based Programming + Inheritance + Dynamic Binding.

Note: **Purely Object Oriented Programming** languages are those languages in which everything accessible(both predefined types and user defined types and even functions) are objects. Languages like C++ and Java are thus not pure OOP languages because they have primitive data types like int , double which are not objects. Example of Pure OOP languages are python, ruby, dart. Etc.

Key Points

1. OOP treats data as a critical element.
2. Emphasis is on data rather than procedure.
3. Decomposition of the problem into simpler modules.
4. Doesn’t allow data to freely flow in the entire system, i.e. localized control flow.
5. Programs are divided into classes and their objects.
6. Data is hidden/protected from external functions.
7. Objects may communicate with each other by functions.

8. It is an Bottom Up Approach. A bottom-up approach begins with low level design or development and ends with high level design.

Advantages of OOP

- *Real World Model & Applications*: It models the real world very well.
- *Maintainability*: With OOP, programs are easy to test, debug, understand and maintain , thus complex software programs are easy to manage.
- *Data Security*: Principle of Data Hiding helps to build secure programs that cannot be invaded by code in other parts of programs.
- *Reusability*: OOP offers code reusability. Already created classes or their features can be reused using inheritance without having to write them again.
- *Fast Development*: OOP facilitates the quick development of programs where parallel development of classes and partition of work is possible and thus higher productivity.
- *Scalability*: Object Oriented Systems can be easily upgraded from small to large systems.

Disadvantages of OOP

1. **Larger program size:** Object-oriented programs typically involve more lines of code than procedural programs.
2. **Slower programs:** Object-oriented programs are typically slower than procedurebased programs, as they typically require more instructions to be executed.
3. **Complex Programs and High Skills:** It is very complex to create programs based on the interaction of objects. Some of the key programming techniques, such as inheritance and polymorphism, can be a big challenging to comprehend initially.

Features of OOP

1. **Data Abstraction**
2. **Data Encapsulation**
3. **Inheritance**
4. **Polymorphism**
5. **Dynamic Binding**
6. **Message Passing**

Applications of OOP

- Real-time systems
- Simulation and modeling
- Object-oriented databases
- Hypertext, hypermedia and expertext
- Al and expert systems
- Neural networks and parallel programming
- Decision support and office automation systems
- CIM/CAM/CAD systems

COMPARISON BETWEEN POP & OOP

BASIS	POP	OOP
Basic	Procedure/Structure oriented .	Object oriented.
Approach	Top-down.	Bottom-up.

Basis	Main focus is on "how to get the task done" i.e. on the procedure or structure of a program .	Main focus is on 'data security'. Hence, only objects are permitted to access the entities of a class.
Division	Large program is divided into units called functions.	Entire program is divided into objects.
Entity accessing mode	No access specifier observed.	Access specifier are "public", "private", "protected".
Overloading or Polymorphism	Neither it overload functions nor operators.	It overloads functions, constructors, and operators.
Inheritance	There is no provision of inheritance.	Inheritance achieved in three modes public private and protected.
Data hiding & security	There is no proper way of hiding the data, so data is insecure	Data is hidden in three modes public, private, and protected. hence data security increases.
Data sharing	Global data is shared among the functions in the program.	Data is shared among the objects through the member functions.
Friend functions or friend classes	No concept of friend function.	Classes or function can become a friend of another class with the keyword "friend". Note: "friend" keyword is used only in C++
Virtual classes or virtual function	No concept of virtual classes .	Concept of virtual function appear during inheritance.
Example	C, VB, FORTRAN, Pascal	C++, Java, C#, Python.

Class & Object

Class
"Class is an user-defined data type, which holds its own data members (or instance variables) and member functions (or methods), which can be accessed and used by creating an instance of that class."

A class is like a blueprint for an object. The primary purpose of the class is to store data and information. Data members & member functions of a class define the properties & behavior of the objects in a class.

Note: *"Memory space for objects is allocated when they are declared and not when class is specified"*.
This statement is only partly true. Actually, the member functions are created and placed in the memory space only once when they are defined as a part of class specification. Thus only space for data members is allocated separately every time object is declared (and memory for code of member functions gets already allocated only once for one class).

Note: Special Characteristics of member functions of a class :
1. Several different classes can use the same function name.
2. Member functions can access the private data of the class. A non-member function cannot do so (except friend functions)
3. A member function can call another member function directly without using dot (.) operator.
4. Member functions defined inside class body are '***inline***' by default. Although , member functions which are defined outside class body can also be made inline by prefixing word inline.

Object
" An Object is an identifiable entity with some characteristics and behavior. An Object is an instance of a Class. When a class is defined, no memory is allocated but when it is instantiated (i.e. an object is created) memory is allocated."
A class is the blueprint of the object, but also, we can say the implementation of the class is the object. The class is not visible to the world, but the object is.

Object	Class
A real-world entity which is an instance of a class	A class is basically a template or a blueprint within which objects can be created
An object acts like a variable of the class	Binds methods and data together into a single unit
An object is a physical entity	A class is a logical entity
Objects take memory space when they are created	A class does not take memory space when created
Objects can be declared as and when required	Classes are declared just once

Real World Example of Class & Object
Consider an *ATM*. An ATM is a class. It's machine which is pretty much useless until you insert your debit card. After you insert your debit card, the machine has information about you and your bank account and the balance in it, so at this point it is an object. You have used a class and created an object, now you can perform operations on it like withdrawal of money or checking you balance or getting statement of your account, these operations will be methods belonging to that class (ATM) but you cannot use them until you create an object out of it. And when you did perform whatever operation you wanted to perform and clicked exit/cancel and removed your card, you just destroyed the object. Now it is not an object, it has methods and all (the functions an ATM can perform) but you cannot use them until you insert your card again and create an object.

Local Class:
- Classes which are defined and used inside a function or a block are known as local class.
- A local class type name can only be used in the enclosing function.
- All the functions of Local classes must be defined inside the class body only.

- A Local class cannot contain static data members. It may contain static functions though.
- Local classes can use global variables (declared above the function) and static variables declared inside the function. Non-static variables of the enclosing function are not accessible inside local classes.
- Also, local classes can access other local classes of same function.
- Enclosing function cannot access the private members of a local class. However, we can achieve this by declaring the enclosing function as a friend.

Nested Class or Inner Class:

Class inside a class is called **nested class**. It is another way of inheriting properties of one class into another.

A nested class is a member and as such has the same access rights as any other member.
The members of an enclosing class have no special access to members of a nested class. The usual access rules shall be obeyed.

Nested Objects:

A class can contain objects of other classes as its members and such objects are known as **member objects** or **nested objects**.This kind of relationship is known as **Containership** or **Nesting**.

A nested object is different from an independent object as it is created in two stages.
First, the member objects are created using their respective constructors and then the other 'ordinary' members are created.
This means constructors of all member objects should be called before its own constructor body is executed which can be accomplished by **Initialization List** inside constructor of nested class.

Example :

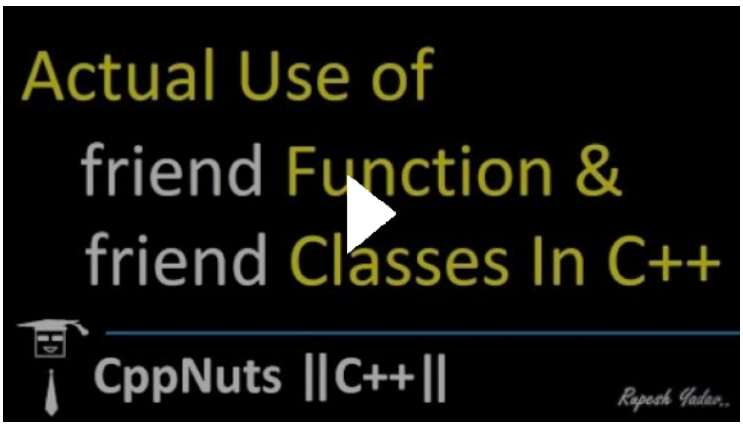
```
class alpha {....};
class beta {....};

 class gamma
 {
        .....
        alpha a;          // a is object of alpha
        beta b;           // b is object of beta
   public:
        gamma(arglist): a(arglist1), b(arglist2)
        {
                  // constructor body
        }
 };
```

Friend Function

Lecture 12 Friend Function in C++ Part 1Hindi (*Watch all 6 Videos*)

Friend Function | Friend Class In C++

Actual Use Of friend Function And Classes In C++



Friend Functions

A non-member function which can access private/protected/public members(data & member functions) of classes in which it is declared as friend is called as *Friend Function*.

Although it should be not be the case that a non-member function getting access of private members of a class as **Data Hiding** will be at stake. Hence friend functions should be used only if absolutely necessary.

Still, it is helpful in cases such as when we need a function which is to be used in more than 1 class but it should be able to access private data, thus implementing **Reusability of Code** by declaring it as friend in all the classes which require the function.

It is also helpful in operator overloading when *istream* or *ostream* objects (cin & cout) are to be overloaded or passed as arguments as they have different class.

A friend function possesses certain special characteristics:

- It is not in the scope of the class to which it has been declared as friend.
  Since it is not in the scope of the class, it cannot be called using the object of that class.
- It can be invoked like a normal function without the help of any object as it is not member function of any class.
- Since friend functions are not member functions of a class, hence they cannot access private members **directly** (without dot operator with an object), hence a reference to an object of class need to be passed as argument to the friend function.
- It does not make difference if they are declared friend in private or public section of class as they are not member functions.

Example)

```
class A{
    int x = 10;
    friend void fun(A, B);
};

class B{
    int y = 10;
    friend void fun(A, B);
};

void fun(A &a, B &b)   // Function Declaration does not require membership label(A:: or B::)
{ cout << a.x << "  " << b.y << endl; }
```

Note: Friend Class

If we want all member functions of one class A as the friend functions of another class B, instead of declaring each function as friend in B, we can declare class A itself as friend In class B.

***Note*:** If a function(or class) is friend to a base class then it does not imply that it is also friend to the child class i.e. **friendship is not inherited**.

Note: **Friendship is not mutual**. If class A is a friend of B, then B doesn't become a friend of A automatically.

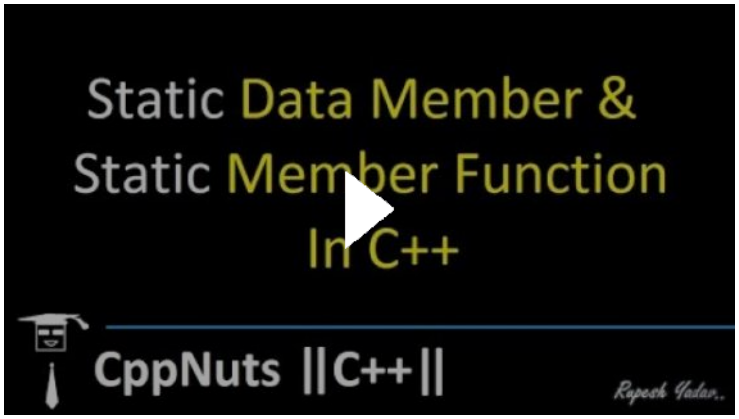
Differences between Friend Function & Friend Class

BASIS FOR COMPARISON	FRIEND FUNCTION	FRIEND CLASS
Definition	It is a function used with a friend keyword to grant a non-member function access to the private members of a class.	It is a class used with a friend keyword to access the private members of another class.
Forward declaration	Must be used.	Not mandatory.
Use	A friend function can be used in some situation of operator overloading.	A friend class can be used when a class is created on the top of another class.

Static Members

Saturday, October 17, 2020 5:12 PM

Static Members



- ***Static Member Variable***
  - Declared inside the class body
  - Also known as class member variable
  - They must be defined outside the class using membership label i.e. they get memory location just like member functions at the time of class specification only. Thus static members will exist even when there is no object of class declared.
  - Static member variable does not belong to any object, but to the whole class. Also thus they don't have ***this*** pointer.
  - There will be only one copy of static member variable for the whole class i.e. all objects of the class will have static members referring to same memory address. Thus any change in value of static members(directly if public or using member function) by an object will be reflected to all the objects of the class.
  - They can also be referred with class name using membership label (class-name::) i.e. they can be used without declaring any object of class.

- ***Static Member Function***
  - They are qualified with the keyword static.
  - They are also called class functions .
  - They can be ***invoked with or without object*** .
  - They can only access other static members(variables and functions) of the class and not non-static ones.

Note: Non-static or normal functions can access both static & non-static members of class but using object only while static member functions can access only static member variables by both using object or membership label (class-name::).

Note: Class with all members as static are known as "*pure static classes*" or "*monostates*" and they

are allocated memory only once (since all members are static and hence no object is required).

Const Objects

Objects which do not modify its data members should be declared constant so that accidental modification of data members can be avoided/prevented.

We may create and use constant objects using **const** keyword before object declaration.

For example, we may create X as a constant object of the class A as follows:
Const A X(m, n); // Object X is constant

Constant Objects cannot modify values of its data members. Any attempt to modify the values of m and n will generate compile-time error.

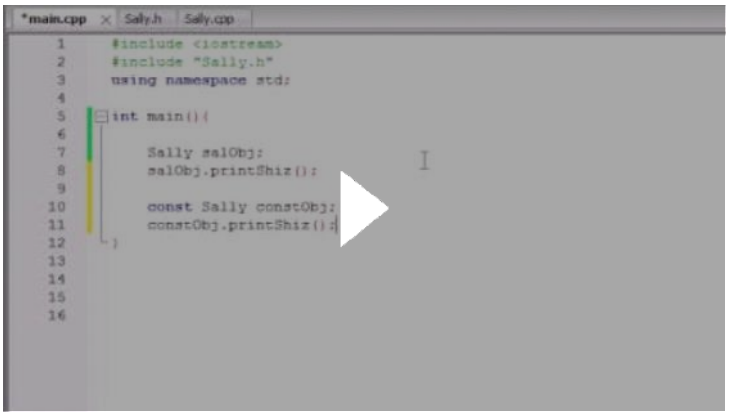
A constant object can only call **const** member functions.

Const Member functions are those functions which cannot alter any data of the class inside their body.
They are postfixed by keyword const after argument list and before opening braces.
Syntax: return-type function-name (argument list) const { … }

Note: Const objects can call only const member functions and cannot call non-const member functions but non-const (normal) objects can call both non-const and const member functions. Whenever const Objects try to invoke non-const member functions , the compiler generates errors.

Note: **Const Data Members** need to be initialized at the same time of their declaration. So assignment operation inside constructor body will not work as it will show constant data being modified. Hence **Initializer List** with Constructor has to be used.


const Member
Function In C++
CppNuts ||C++||
Rupesh Yadav..

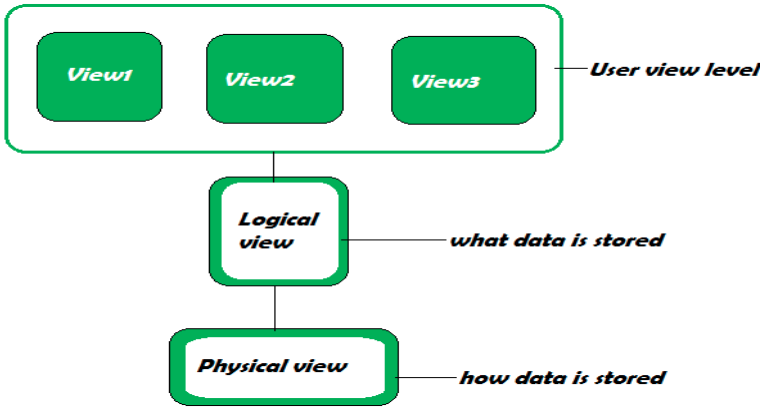

*main.cpp
Sally.h
Sally.cpp
#include <iostream>
#include "Sally.h"
using namespace std;
int main(){
Sally salObj;
salObj.printShiz();
const Sally constObj;
constObj.printShiz();
}

Abstraction & Encapsulation

Data Abstraction

"Abstraction means displaying only essential information and hiding the details. Data abstraction refers to providing only essential information about the data to the outside world, hiding the background details or implementation."

Abstraction refers to the act of representing important and special features without including the background details or explanation about that feature. Data abstraction simplifies database design. Since classes implement Data Abstraction, hence classes are called *ABSTRACT DATATYPE (ADT)*.



a. **Physical Level:**
   It describes how the records are stored, which are often hidden from the user. It can be described with the phrase, "block of storage."
b. **Logical Level:**
   It describes data stored in the database and the relationships between the data. The programmers generally work at this level as they are aware of the functions needed to maintain the relationships between the data.
c. **View Level:**
   Application programs hide details of data types and information for security purposes. This level is generally implemented with the help of GUI, and details that are meant for the user are shown.

A) How to achieve data abstraction in C++ ?

1. **Abstraction using Classes:** We can implement Abstraction in C++ using classes. The class helps us to group data members and member functions using available access specifiers. A Class can decide which data member will be visible to the outside world and which is not.
   Note: Abstraction is mostly done by *interfaces* rather than *abstract classes*.

2. **Abstraction in Header files:** One more type of abstraction in C++ can be header files. For example, consider the pow() method present in math.h header file. Whenever we need to calculate the power of a number, we simply call the function pow() present in the math.h header file and pass the numbers as arguments without knowing the underlying algorithm according to which the function is actually calculating the power of numbers.

3. **Abstraction Using Access Specifier**: The members that defines the internal implementation can be marked as private in a class. And the important information needed to be given to the outside world can be marked as public. And these public members can access the private members as they are inside the class.

Advantages or Need of Data Abstraction:

- Helps the user to avoid writing the low level code
- Avoids code duplication and increases reusability.
- Can change internal implementation of class independently without affecting the user.
- Helps to increase security of an application or program as only important details are provided to the user.

Real world example of Data Abstraction is *ATM Machine*: All are performing operations on the ATM machine like cash withdrawal, money transfer, retrieve mini-statement…etc. but we can't know internal details about ATM.

Encapsulation

"Encapsulation is defined as wrapping up of data and information under a single unit. In Object-Oriented Programming, Encapsulation is defined as binding together the data and the functions that manipulate them."

Encapsulation also leads to data abstraction and data hiding (as using encapsulation also hides the data). This concept is often used to hide the internal state representation of an object from the outside.

Encapsulation => Data Abstraction + Data Hiding

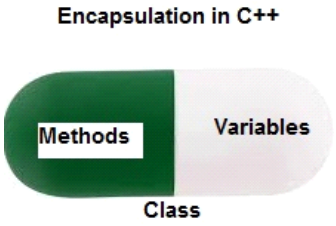
Q) How to achieve Encapsulation in C++ ?

In C++ , it can be implemented using Class Body and access specifiers. If you are creating class, you are doing encapsulation . Access specifiers plays an important role in implementing encapsulation in C++.

The process of implementing encapsulation can be sub-divided into two steps:

1. The data members should be labeled as private using the private access specifiers.
2. The member function which manipulates the data members should be labeled as public using the public access specifier.

Real world example of Encapsulation is *medicine capsule*. All medicine ingredients are encapsulated inside a single capsule.



Need or Advantages of Encapsulation

- Encapsulation protects an object from unwanted access by clients.
- Encapsulation allows access to a level without revealing the complex details below that level.
- It reduces human errors and Makes the application easier to understand.
- Simplifies the maintenance of the application by organizing the code better.

Differences between Abstraction & Encapsulation

S.NO	Abstraction	Encapsulation
1.	Abstraction is the process or method of gaining the information.	While encapsulation is the process or method to contain the information.
2.	In abstraction, problems are solved at the design or interface level.	While in encapsulation, problems are solved at the implementation level.
3.	Abstraction is the method of hiding the unwanted information.	Whereas encapsulation is a method to hide the data in a single entity or unit along with a method to protect information from outside.
4.	We can implement abstraction using abstract class and interfaces.	Whereas encapsulation can be implemented using by access modifier i.e. private, protected and public.
5.	In abstraction, implementation complexities are hidden using abstract classes and interfaces.	While in encapsulation, the data is hidden using methods of getters and setters.
6.	The objects that help to perform abstraction are encapsulated.	Whereas the objects that result in encapsulation need not be abstracted.

Polymorphism

"The word polymorphism means having many forms. polymorphism is the ability of a message to be displayed in more than one form."
Polymorphism is the ability of data to be processed in more than one form. It allows the performance of the same task in various ways. It consists of method overloading and method overriding, i.e., writing the method once and performing a number of tasks using the same method name.

Real world example of polymorphism can be a *girl*. She can be a daughter, mother, sister and and in all a human being.
Another example can be a *mobile phone*. The same mobile phone is used to take calls, click pictures & videos, run calculator and other applications, etc.

Advantages of Polymorphism:
- It helps the programmer to reuse the codes, i.e., classes once written, tested and implemented can be reused as required. Saves a lot of time.
- Single variable can be used to store multiple data types.
- Easy to debug the codes.

Disadvantages of Polymorphism:
- Run time polymorphism can lead to the performance issue as machine needs to decide which method or variable to invoke so it basically degrades the performances as decisions are taken at run time.
- Polymorphism reduces the readability of the program. One needs to identify the runtime behavior of the program to identify actual execution time.

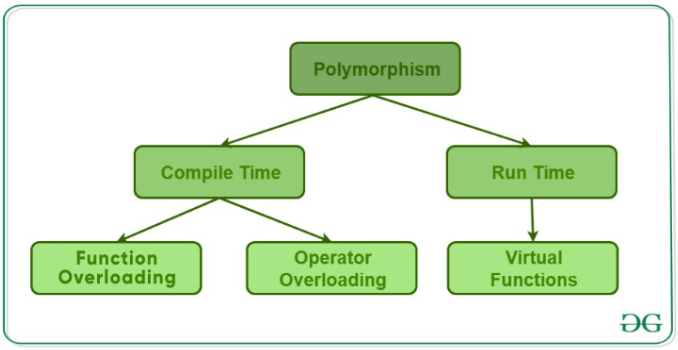
Polymorphism is extensively used in implementing inheritance. An operation may exhibit different behaviors in different instances. The behavior depends upon the types of data used in the operation.

In C++ polymorphism is mainly divided into two types:

1. **Compile time Polymorphism**
   This type of polymorphism is achieved by function overloading or operator overloading.
   - ***Function Overloading***: Function overloading is using a single function name to perform different types of tasks. When there are multiple functions with same name but different parameters then these functions are said to be overloaded. Functions can be overloaded by change in number of arguments or/and change in type of arguments.
   - ***Operator Overloading***: The process of making an operator to exhibit different behaviors in different instances is known as operator overloading. C++ also provide option to overload operators. For example, we can make the operator ('+') for string class to concatenate two strings.

   The overloaded member functions are '*selected*' for invoking by matching arguments, both type and number. This information is known to the compiler at the *compile time* and therefore, compiler is able to select the appropriate function for a particular call at the compile time itself. This is also called ***early binding*** or ***static binding*** or ***static linking***. Early binding simply means an object is bound to its function call at compile time.

2. **Runtime Polymorphism**
   This type of polymorphism is achieved by Function overriding along with virtual functions.
   Function Overriding: It occurs when a derived class has a definition for one of the member functions of the base class. That base function is said to be **overridden**.
   C++ supports a mechanism known as ***virtual functions*** to achieve run-time polymorphism i.e. selecting appropriate member function at run time.
   At run time, when it is known what class objects are under consideration, the appropriate version of the function is invoked. Since the function is linked with a particular class much later after the compilation, this process is known as ***late binding***. It is also known as ***dynamic binding*** because the selection of the appropriate function is done dynamically at run time.
   Dynamic binding is one the powerful features of C++. This requires the use of pointers to objects.



Differences between Polymorphism & Inheritance

S.NO	Inheritance	Polymorphism
1.	Inheritance is one in which a new class is created (derived class) that inherits the features from the already existing class(Base class).	Whereas polymorphism is that which can be defined in multiple forms.
2.	It is basically applied to classes.	Whereas it is basically applied to functions or methods.
3.	Inheritance supports the concept of reusability and reduces code length in object-oriented programming.	Polymorphism allows the object to decide which form of the function to implement at compile-time (overloading) as well as run-time (overriding).
4.	Inheritance can be single, hybrid, multiple, hierarchical and multilevel inheritance.	Whereas it can be compiled-time polymorphism (overload) as well as run-time polymorphism (overriding).
5.	It is used in pattern designing.	While it is also used in pattern designing.

Dynamic Binding & Message Passing

Dynamic Binding:

Binding refers to the linking of a procedure call to the code to be executed in response to the call. In Simple words, connecting a method call to the method body is known as binding.
There are two types of binding:

- Static Binding (also known as Early Binding): Type of object is determined at compile time.

- Dynamic Binding (also known as Late Binding): Type of object is determined at run-time.

  Dynamic binding (also known as late binding) means that the code to be executed in response to a given procedure/function call is not known until the run-time. It is associated with polymorphism and inheritance. A function associated with a polymorphic reference depends on the dynamic type of that reference. C++ has *virtual functions* to support this.

Message Passing:

Objects communicate with one another by sending and receiving information to each other. A message for an object is a request for execution of a procedure and therefore will invoke a function in the receiving object that generates the desired results. Message passing involves specifying the name of the object, the name of the function and the information to be sent. An object-oriented program consists of a set of objects that communicate with each other.

Message Passing involves the following steps :
1. Creating classes that define objects and their behavior
2. Creating objects from class definitions
3. Establishing communication among objects

Objects communicate with one another by sending and receiving information much the same way as people pass messages to one another. The concept of message passing makes it easier to talk about building systems that directly model or simulate their real-world counterparts. A message for an object is a request for execution of a procedure, and therefore will invoke a function (procedure) in the receiving object that generates the desired result. Message passing involves specifying the name of the object, the name of the function (message) and the information to be sent.

Inheritance

"The capability of a class (child class) to derive properties and characteristics from another class (parent class) is called Inheritance."
When we create a class, we do not need to write all the properties and functions again and again, as these can be inherited from another class which possesses it. Inheritance allows the user to reuse the code whenever possible and reduce its redundancy.

Real world example of inheritance can be *humans*. We inherit certain properties from the class 'Human' such as the ability to speak, breathe, eat, drink, etc.
We can also take the example of *cars*. The class 'Car' inherits its properties from the class 'Automobiles' which inherits some of its properties from another class 'Vehicles'.

Limitations or Disadvantages of inheritance

- Increases the time and effort required to execute a program as it requires jumping back and forth between different classes
- Any modifications to the program would require changes both in the parent as well as the child class
- Needs careful implementation else would lead to incorrect results
- Inherited functions work slower than normal function as there is indirection.
- Often, data members in the base class are left unused which may lead to memory wastage.
- Inheritance increases the coupling between base class and derived class. A change in base class will affect all the child classes.

Advantages or Need of Inheritance

- Inheritance promotes reusability. When a class inherits or derives another class, it can access all the functionality of inherited class.
- Reusability enhanced reliability. The base class code will be already tested and debugged.
- As the existing code is reused, it leads to less development and maintenance costs.
- Inheritance makes the sub classes follow a standard interface.
- Inheritance helps to reduce code redundancy and supports code extensibility.
- Inheritance facilitates creation of class libraries.

ACCESS SPECIFIER / ACCESS MODIFIER

Access modifiers are used to implement an important feature of Object-Oriented Programming known as Data Hiding.

Access Modifiers or Access Specifiers in a class are used to set the accessibility of the class members. That is, it sets some restrictions on the class members not to get directly accessed by the outside functions.

There are 3 types of access modifiers in C++:

1. **Public:** All the class members declared under public will be available to everyone. The data members and member functions declared public can be accessed by other classes too. The public members of a class can be accessed from anywhere in the program using the direct member access operator (.) with the object of that class.
2. **Private:** The class members declared as private can be accessed only by the functions inside the class. They are not allowed to be accessed directly by any object or function outside the class. Only the member functions or the friend functions are allowed to access the private data members of a class.
3. **Protected:** Protected access modifier is similar to that of private access modifiers, the difference is that the class member declared as Protected are inaccessible outside the class but they can be accessed by any subclass(derived class) of that class.

INHERITANCE

- **Sub Class:** The class that inherits properties from another class is called **Derived Class** or Sub class. It can be defined by specifying its relationship with the base class(es) in addition to its own details.
- **Super Class:** The class whose properties are inherited by sub class is called **Base Class** or Super class.
- **Reusability:** Inheritance supports the concept of "reusability", i.e. when we want to create a new class and there is already a

class that includes some of the code that we want, we can derive our new class from the existing class. By doing this, we are reusing the fields and methods of the existing class.

Definition: Mechanism of deriving a new class(known as child class) from an old one(known as base class) is called inheritance or derivation. The derived class inherits some or all of the traits from the base class.

Modes of Inheritance or Visibility Mode

4. **Public mode**: If we derive a sub class from a public base class. Then the public member of the base class will become public in the derived class and protected members of the base class will become protected in derived class.
5. **Protected mode**: If we derive a sub class from a Protected base class. Then both public member and protected members of the base class will become protected in derived class.
6. **Private mode**: If we derive a sub class from a Private base class. Then both public member and protected members of the base class will become Private in derived class.

Note : **Private** data members of base class are although **available** in child class but **not accessible directly**. They can be accessed in derived class using member functions of base class. On the other hand, **protected** data members of base class are **accessible directly** in derived class' member functions(not using objects).

Note : If no visibility mode is specified while inheriting the base class, then by **default** it is inherited in **private** mode.

Note : Private & Protected data members of a class can be accessed by the following:

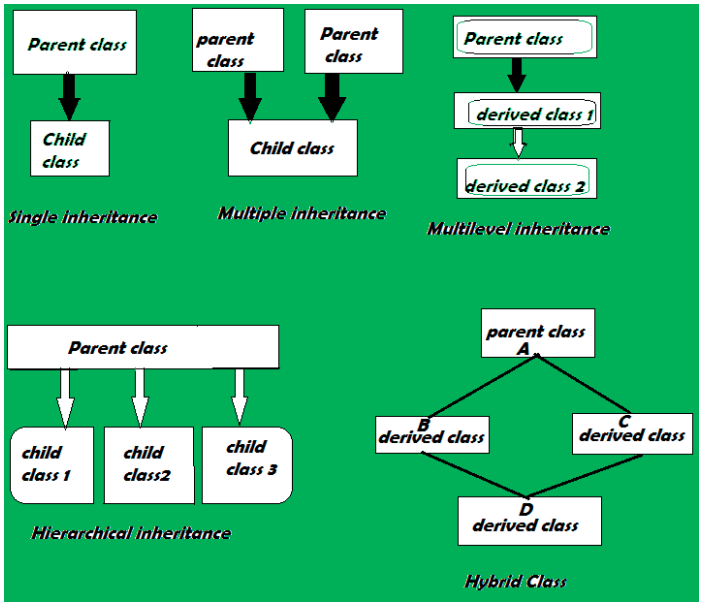
a. Member functions of the class
b. Member functions of derived class
   (they can **directly access protected** data members only but can **indirectly access private** data members using member functions of base class)
c. Friend functions
d. Member functions of friend class
e. Static Member functions { only if data member is **static** }

Public data members of a class can be accessed by the above all plus the following:

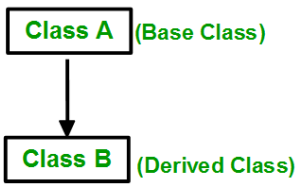
a. Objects of the class using dot operator(.)
b. Pointer to object of class using arrow operator (->)
c. Objects of Derived class (only if Mode of inheritance is **public**) { using dot operator(.) }
d. Class Membership Label (ClassName::) { only if data member is declared as **static** }

Base class member access specifier	Type of Inheritence		
	Public	Protected	Private
Public	Public	Protected	Private
Protected	Protected	Protected	Private
Private	Not accessible (Hidden)	Not accessible (Hidden)	Not accessible (Hidden)

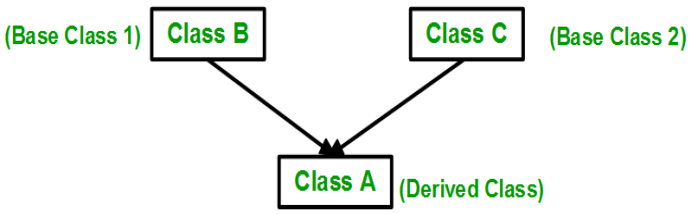
Types of Inheritance



1. **Single Level Inheritance**: In single inheritance, a class is allowed to inherit from only one class. i.e. one sub class is inherited by one base class only.



2. **Multiple Inheritance:** Multiple Inheritance is a feature of C++ where a class can inherit from more than one classes. i.e one sub class is inherited from more than one base classes.
It allows us to combine the features of several existing classes as a starting point for defining new classes.



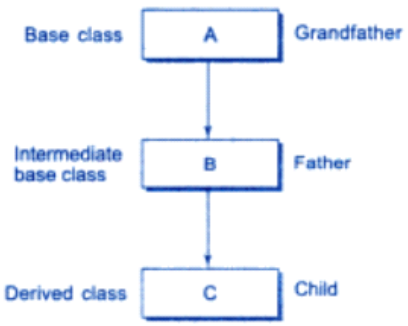
Syntax of a derived class with multiple base classes is using following where base classes are separated by commas.

```
class D: visibility B-1, visibility B-2 ...
{
      .....
      .....(Body of D)
      .....
};
```

Note: It may happen that the base classes which are inherited in child class using multiple inheritance may have function with same name and arguments. It will cause an **ambiguity** in such a case that which base class' function will be called. To resolve the ambiguity, membership label (baseclass name + scope resolution operator) should be used.

```
class M
{
  public:
      void display(void)
      {
            cout << "Class M\n";
      }
};

class N
{
  public:
      void display(void)
      {
            cout << "Class N\n";
      }
};

class P : public M, public N


{
  public:
      void display(void)        // overrides display() of M and N
      {
            M :: display();
      }
};
```

3. **Multilevel Inheritance**: In this type of inheritance, a derived class is derived from another derived class.

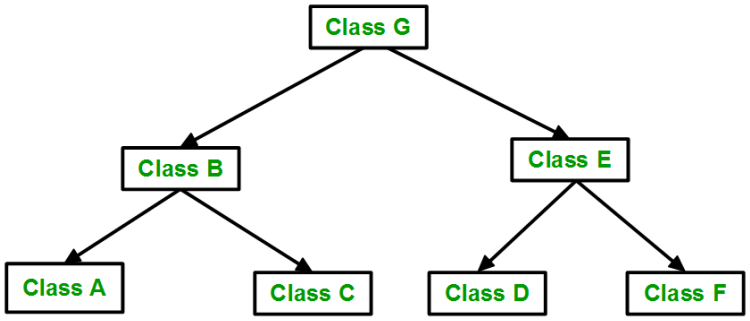


The class A serves as a base class for the derived class B which in turn serves as base class for the derives class C. The class B is known as **Intermediate Base Class** since it provides a link for the inheritance between A and C. The chain ABC is known as **Inheritance Path**.

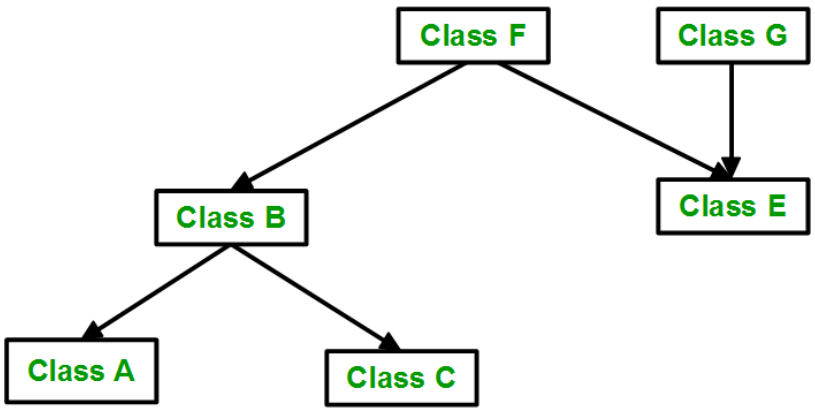
Multiple Inheritance	Multilevel Inheritance
Multiple inheritance comes into picture when a class inherits more than one base class	Multilevel inheritance means a class inherits from another class which itself is a subclass of some other base class
Example: A class defining a child inherits from two base classes Mother and Father	Example: A class describing a sports car will inherit from a base class Car which inturn inherits another class Vehicle

4. **Hierarchical Inheritance**: In this type of inheritance, more than one sub class is inherited from a single base

class. i.e. more than one derived class is created from a single base class. Hence it can be used to support the hierarchial design of a program i.e. certain features of one level are shared by many others below that level.

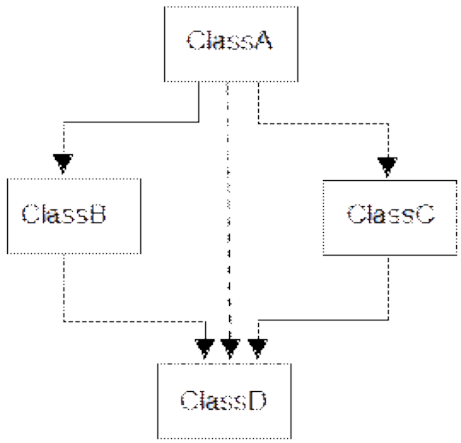


5. **Hybrid (Virtual) Inheritance**: Hybrid Inheritance is implemented by combining more than one type of inheritance. For example: Combining Multilevel inheritance and Multiple Inheritance.
Below image shows the combination of hierarchical and multiple inheritance:



6. **Multipath inheritance** (Special Case of Hybrid Inheritance) :
A derived class with two base classes and these two base classes have one common base class is called multipath inheritance. It is a hybrid inheritance with all three major kinds of inheritance, namely multiple inheritance, hierarchial inheritance and multilevel inheritance are involved.

   Here, classA is *indirect base class* of classD and classB & classC are direc base classes of classD.



An ambiguity can arise in this type of inheritance and the problem is known as **Diamond Problem** or **The Diamond of Death**. In the following example class D will get 2 copies of data members & member functions of class A (one through B and one through C). Solution to this problem is inheriting A in B and C with '***virtual***' keyword making B & C virtual base classes for class D. This is known as ***virtual inheritance*** *or* ***disinheritance***. The base class A is known as **virtual base class**. Note that *virtual* and *public* keywords (in syntax of inheriting)

can appear in any order.

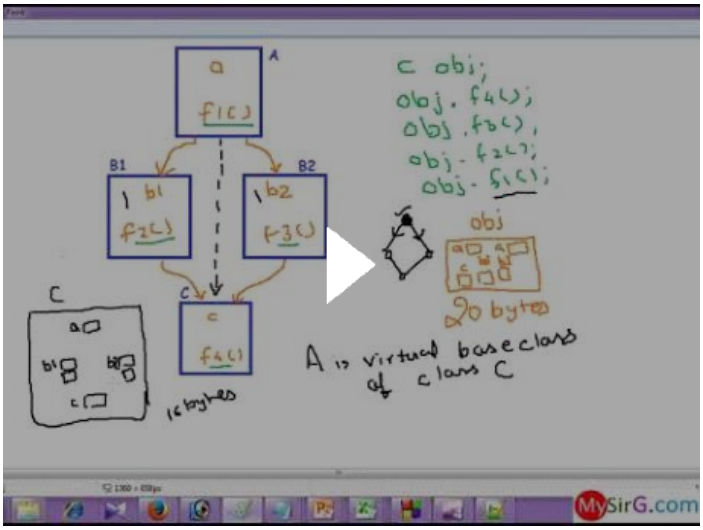
Virtual base classes are used in virtual inheritance in a way of preventing multiple *instances* of a given class appearing in an inheritance hierarchy when using multiple inheritances. Virtual base classes offer a way to save space and avoid ambiguities in class hierarchies that use multiple inheritances. When a base class is specified as a virtual base, it can act as an indirect base more than once without duplication of its data members. A single copy of its data members is shared by all the base classes that use virtual base.

Example :

```
class Person {
   public:
    Person(int x)  { cout << "Person::Person(int ) called" << endl;   }
};

class Faculty : virtual public Person {
   public:
    Faculty(int x):Person(x)   {
       cout<<"Faculty::Faculty(int ) called"<< endl;
    }
};

class Student : virtual public Person {
   public:
    Student(int x):Person(x) {
        cout<<"Student::Student(int ) called"<< endl;
    }
};

class TA : public Faculty, public Student  {
    public:
    TA(int x):Student(x), Faculty(x)   {
        cout<<"TA::TA(int ) called"<< endl;
    }
};
```

Hybrid Inheritance Diamond Problem in Hindi

Virtual base class Hindi



Constructors & Destructors in Inheritance

Lecture 14 Constructor and Destructor in Inheritance in C++ Part 1 Hindi

As long as there are only base class constructor which does not take any arguments(only default constructors), the derived class need not have a constructor function.
However, if any base class contains a constructor with one or more arguments (parameterized or copy), then it is **mandatory** for the derived class to have a constructor and pass the arguments to the base class constructor.

Order of Execution of Constructors
In **single inheritance**, the base constructor is executed first and then the constructor in the derived class is executed. In case of **multiple inheritance**, the base classes are constructed in the order in which they appear in the declaration of the derived class. Similarly, in a **multilevel inheritance**, the constructors will executed in the order of inheritance.

Contructors for Virtual Base Classes
The constructors for virtual base classes are invoked before any non-virtual base classes. If there are multiple virtual base classes, they are invoked in the order in which they are declared. Any non-virtual bases are then constructed before the derived class constructor is executed.

Example)

Method of inheritance	*Order of execution*

Class B: public A
{
};
	A( ) ; base constructor
	B( ) ; derived constructor

class A : public B, public C
{
};
	B( ) ; base(first)
	C( ) ; base(second)
	A( ) ; derived

class A : public B, virtual public C
{
};
	C( ) ; virtual base
	B( ) ; ordinary base
	A( ) ; derived

Order of Execution of Destructors

Destructors are executed in reverse order to what constructors were executed i.e. destructor of derived class will be executed first and then the destructor of base class will be executed.

Q) Why the base class's constructor is called on creating an object of derived class?

A) When a class is inherited from other,The data members and member functions of base class comes automatically in derived class based on the access specifier but the definition of these members exists in base class only. So when we create an object of derived class, all of the members of derived class must be initialized but the inherited members in derived class can only be initialized by the base class's constructor as the definition of these members exists in base class only. This is why the constructor of base class is executed first to initialize all the inherited members.

Note: Although base class' constructor is **executed** first before derived class' constructor but derived class' constructor is **called** first before base class constructor (using the object of derived class). When derived class constructor is called, this constructor itself calls the constructor of base class (present in its initializer list) before executing it's own body. Hence, **constructors & destructors cannot be inherited** but only called implicitly or explicitly by the child constructor.

Historically constructors could not be inherited in the C++03 standard. You needed to inherit them manually one by one by calling base implementation on your own.

Constructor Inheritance in C++11

C++11 standard, there is can be constructor inheritance with **using** keyword.

```
class A
{
   public:
      explicit A(int x) {}
};

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

This is all or nothing - you cannot inherit only some constructors, if you write this, you inherit all of them. To inherit only selected ones you need to write the individual constructors manually and call the base constructor as needed from them.

Composition vs Inheritance

S.NO	Inheritance	Composition
1.	In inheritance, we define the class which we are inheriting(super class) and most importantly it cannot be changed at runtime	Whereas in composition we only define a type which we want to use and which can hold its different implementation also it can change at runtime. Hence, Composition is much more flexible than Inheritance.
2.	Here we can only extend one class, in other words	Whereas composition allows to use functionality from

	more than one class can't be extended as java do not support multiple inheritance.	different class.
3.	In inheritance we need parent class in order to test child class.	Composition allows to test the implementation of the classes we are using independent of parent or child class.
4.	Inheritance cannot extend final class.	Whereas composition allows code reuse even from final classes.
5.	It is an is-a relationship.	While it is a has-a relationship.

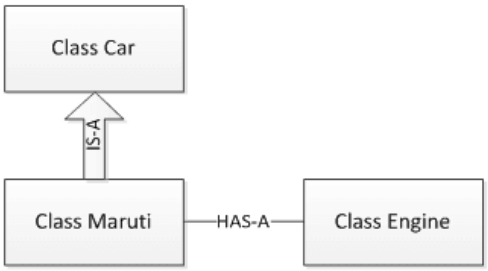
Is A vs Has A relationship => Abdul Bari Video

IS-A Relationship:

- In object-oriented programming, the concept of IS-A is a totally based on Inheritance, which can be of two types Class Inheritance or Interface Inheritance. For example, Apple is a Fruit, Car is a Vehicle etc.
- Inheritance is unidirectional. For example, House is a Building. But Building is not a House.
- It is a key point to note that you can easily identify the IS-A relationship. Wherever you see an extends keyword or implements keyword in a class declaration, then this class is said to have IS-A relationship.

HAS-A Relationship:

- Composition (HAS-A) simply mean the use of instance variables that are references to other objects.
- For example, Maruti has Engine, or House has Bathroom.



Generalization vs Specialization => Abdul Bari Video

Generalization:

Generalization is the process of extracting shared characteristics from two or more classes, and combining them into a generalized superclass. Shared characteristics can be attributes, associations, or methods.

Eg) Circle, square and rectangle are derived classes from parent class shape. Only derived classes exists in real world and shape is just generalized class for those real world objects.

Specialization:

If some new subclasses are created from an existing superclass to do specific job of the superclass, then it is known as specialization.

Eg) Cuboid is a derived class from parent class rectangle. Both rectangle and cuboid exists in real world.

Constructors

Constructors

- A constructor is a *special* member function of the class whose task is to initialize object(s) of its class.
- It is special because its name is the same as the Class name.
- The constructor is automatically called / invoked whenever an object of its associated class is created.
- It is called constructor because it constructs the values of data members (instance variables) of the class.

Characteristics of Constructor

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

Advanced Constructors

- **Virtual constructor** in C++
  The virtual mechanism works only when we have a base class pointer to a derived

class object.
In C++, the constructor cannot be virtual, because when a constructor of a class is executed there is no virtual table in the memory, means no virtual pointer defined yet. So, the constructor should always be non-virtual.
Note: But virtual destructor is possible.

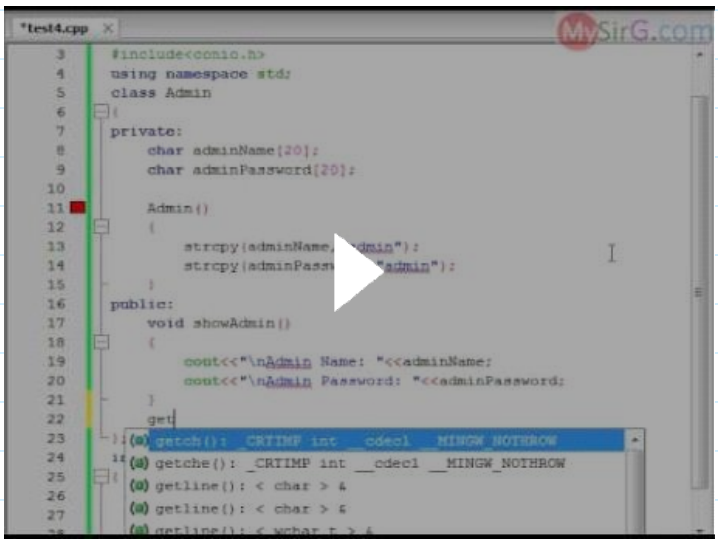
- **Dynamic Constructor** in C++
  When the data members of a class are allocated memory dynamically inside a constructor , it is called **dynamic construction of object** and such a constructor which is used in Dynamic Memory Allocation of data members of class is called Dynamic Constructor.

  Note: If we create a pointer to object and dynamically allocate it memory using new , such a constructor may or may not be Dynamic Constructor because the only condition for a constructor to be dynamic is to dynamically allocate memory to data members of class not dynamically allocate memory to object itself.

  Eg) ClassName *objptr = new ClassName;
  // This need not be Dynamic Constructor

  Eg) class A{
  Int *p;
  A() // This is a DYNAMIC CONSTRUCTOR
  {
  p = new int;
  *p = 0;
  }
  };
  A obj;

- **Private Constructor** in C++
  Article: https://www.geeksforgeeks.org/can-constructor-private-cpp/
  Video:



Types of Constructors

1. ***Default Constructors:*** It is the constructor which doesn't take any argument. It has no parameters. We can both define our own default constructors (to initialize data members with some non-garbage value) or use the default constructor which is provided by C++ compiler implicitly.

Note: Even if we do not define any constructor explicitly, the compiler will automatically provide a default constructor implicitly.(expects no parameters and has an empty body).
For eg) Classname objectname;
This statement of object declaration leads to default constructor provided by C++ compiler to be called implicitly to initialize the object's data members with junk values. However, it is not necessary but it's considered to be the best practice to always define a default constructor.

Note: Whenever we define one or more non-default(parameterized or copy) constructors for a class, a default constructor(without parameters) ***must*** also be explicitly defined as the compiler will not create a default constructor in this case.

Hence *ClassName ObjName;* will give compile time error if we do not write our own default constructor but write only our own parameterized or copy constructor.

Note: A parameterized constructor with all default arguments(in short *default argument constructor*) when called with no arguments are also treated as default constructor. Thus default constructor and default argument constructor cannot be declared simultaneously as it will cause ambiguity for statement such as *Classname ObjName;*

Note: C++ allows even built-in type (primitive types) to have default constructors.
For eg) *cout << int() << endl;* int() is default constructor and cout statement will print 0.

2. ***Parameterized Constructors:*** These constructors can take one or more parameters/arguments.

Typically, these arguments help initialize an object when it is created. When an object is declared in a parameterized constructor, the initial values *must* have to be passed as

arguments to the constructor function. The normal way of object declaration may not work (without calling constructor implicitly/explicitly).

The constructors (parameterized and copy both) can be called **explicitly** or **implicitly**. Only Syntax is different but meaning is same.

The following declaration illustrates the first method:

```
integer int1 = integer(0,100); // explicit call
```

This statement creates an integer object int1 and passes the values 0 and 100 to it. The second is implemented as follows:

```
integer int1(0,100);                    // implicit call
```

This method, sometimes called the shorthand method, is used very often as it is shorter, looks better and is easy to implement.

Note: When we need compiler to avoid implicit call to constructor , we can define the constructor **explicit** by adding explicit keyword in front of constructor name.
Eg) Class A { explicit A() {..} };

Uses of Parameterized constructor:

- It is used to initialize the various data elements of different objects with different values when they are created.
- It is used to overload constructors.

3. ***Copy Constructor:***
   A copy constructor is a member function which initializes an object using another object of the same class. Another Object which is passed in argument must be passed by reference.

A copy constructor has the following general function prototype:

```
ClassName (const ClassName &old_obj);
```

Cases When Copy Constructor is Called

1. When an object of the class is returned by value.
2. When an object of the class is passed (to a function) by value as an argument.
3. When an object is constructed based on another object of the same class.
4. When the compiler generates a temporary object.

It is, however, not guaranteed that a copy constructor will be called in all these cases, because the C++ Standard allows the compiler to optimize the copy away in certain cases.

Default Copy Constructor : If we don't define our own copy constructor, the C++ compiler creates a default copy constructor for each class which does a member-wise copy between objects. The compiler created copy constructor works fine in general (without involvement of pointers). *Default constructor does only* ***shallow copy****.*

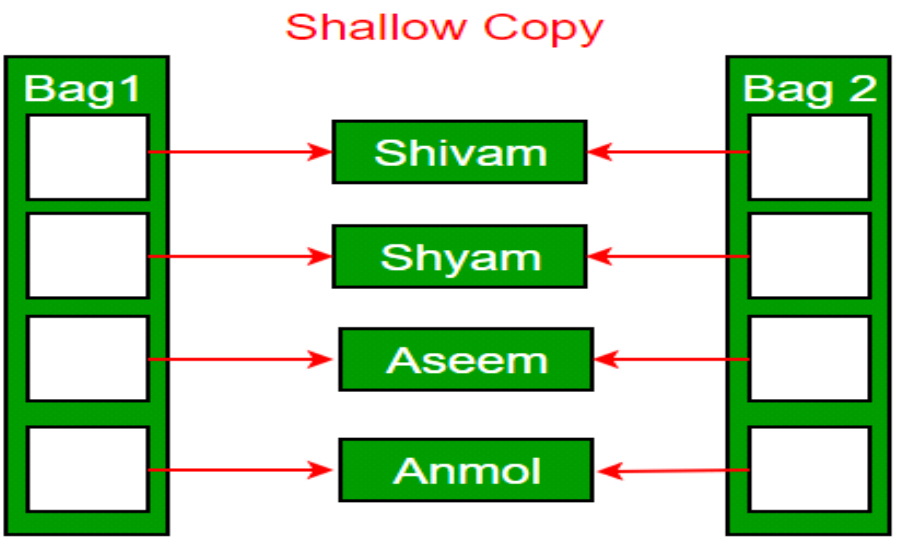
The problem with default copy constructor (and assignment operator) is – When we have members which dynamically gets initialized at run time, default copy constructor copies these members with same address of dynamically allocated memory and not real copy of this memory. Now both the objects points to the same memory and changes in one reflects in another object, Further the main disastrous effect is, when we delete one of this object other object still points to same memory, which will be dangling pointer, and memory leak is also possible problem with this approach. In such cases, we should always write our own copy constructor (and assignment operator).

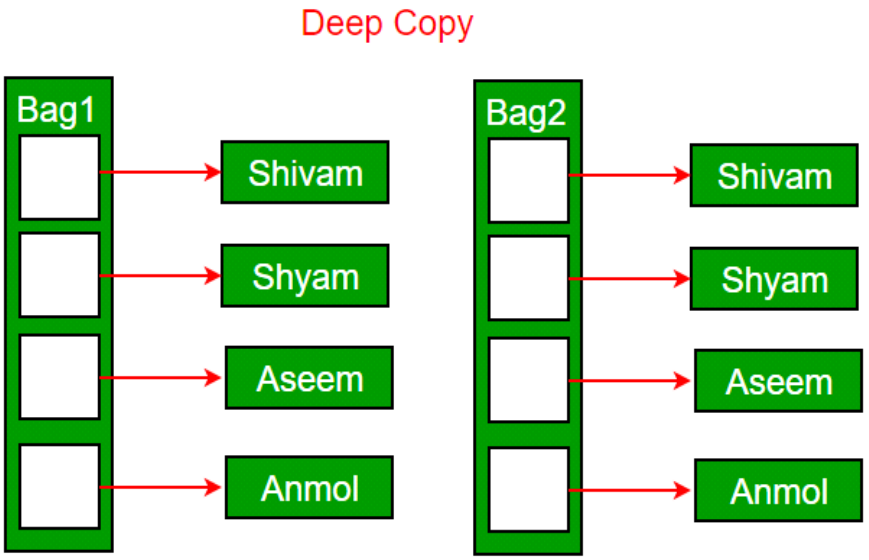
User Defined Copy Constructor : We need to define our own copy constructor only if an object has pointers or any runtime allocation of the resource like file handle, a network connection etc.

Deep copy *is possible only with user defined copy constructor.* In user defined copy constructor, we make sure that pointers (or references) of copied object point to new memory locations.

Lecture 23 Deep Copy and Shallow Copy in C++ Hindi


Deep Copy and Shallow Copy
Lecture 23 (Hindi)
By: Saurabh Shukla
09893194395


Shallow Copy
Bag1
Bag 2
Shivam
Shyam
Aseem
Anmol


Deep Copy
Bag1
Bag2
Shivam
Shyam
Aseem
Anmol
Shivam
Shyam
Aseem
Anmol

Copy constructor vs Assignment Operator

Copy constructor is called when a new object is created from an existing object, as a copy of the existing object. Assignment operator is called when an already initialized object is assigned a new value from another existing object.

```
class Test
{
   public:
   Test() {}
   Test(const Test &t)                          // User Defined Copy Constructor
   {  cout<<"Copy constructor called "<<endl;  }
   Test& operator = (const Test &t)       // Assignment Operator(=) Overloading
   {
      cout<<"Assignment operator called "<<endl;
      return *this;
   }
};
Test t1, t2;
Test t3 = t1;  // ----> Copy Constructor
t2 = t1;         // -----> Assignment Operator
```

Q) Why argument to a copy constructor must be passed as a reference?

A copy constructor is called when an object is passed as argument. Copy constructor itself is a function. If the argument is passed by value, its copy constructor would call itself to copy the actual parameter to formal parameter. This process would go on until the system runs out of memory. So, we should pass it by reference , so that copy constructor does not get invoked. Therefore compiler doesn't allow parameters to be passed by value & thus it will give compile time error.

A) Why argument to a copy constructor should be const?

When we create our own copy constructor, we pass an object by reference and we generally pass it as a const reference. Reason for passing const reference is, we should use const in C++ wherever possible so that objects are not accidentally modified.

Q) Can we make copy constructor private?

Yes, a copy constructor can be made private. When we make a copy constructor private in a class, objects of that class become non-copyable. This is particularly

useful when our class has pointers or dynamically allocated resources. In such situations, we can either write our own copy constructor like above example or make a private copy constructor so that users get compiler errors rather than surprises at runtime.

A) Can copy constructor be virtual?
B) No, Copy constructor cannot be made virtual in C++. (Concept of virtual constructor, be it of any type, is not allowed in C++.)

Note: Though, there is a *virtual copy constructor idiom* for advanced c++. For extra reading: https://www.tutorialspoint.com/virtual-copy-constructor-in-cplusplus

Destructors

A destructor is used to destroy/delete the objects that have been created by a constructor.

Like a constructor, the destructor is a member function whose name is the same as the class name but is preceded by a tilde. A destructor neither takes any argument nor does it return any value.

There can be only one destructor for a class i.e. *destructor overloading* is **not** possible.
Reasons:

- There are no parameters for destructor (nor has a return type).
- There is nothing optional when you are destroying the object.

It will be invoked implicitly by the compiler when the object goes out of scope(upon exit from the program or block or function or delete operator called) to clean up storage that is no longer accessible.

Q) Need of User Defined Destructor?

If we do not write our own destructor in class, compiler creates a ***default destructor*** for us. The default destructor works fine unless we have dynamically allocated memory or pointer in class. When a class contains a pointer to memory allocated in class, we should write a destructor to release memory before the class instance is destroyed. This must be done to avoid memory leak. It is a good practice to declare destructors in a program since it releases memory space for future use.

Whenever **new** is used to make pointer to class object, we should use **delete** to free the memory. This is required because when the pointers to objects go out of scope. a destructor is **not** called implicitly.

A) Can a destructor be virtual?

Yes, In fact, it is always a good idea to make destructors virtual in base class when we have a virtual function.

Deleting a derived class object using a pointer to a base class that has a non-virtual destructor results in undefined behavior. To correct this situation, the base class should be defined with a virtual destructor. For example, following program results in undefined behavior. Making base class destructor virtual guarantees that the object of derived class is destructed properly, i.e., both base class and derived class destructors are called.

As a guideline, any time you have a virtual function in a class, you should immediately add a virtual destructor (even if it does nothing).

Virtual Destructor in C++ Article: https://www.geeksforgeeks.org/virtual-destructor/

Pure Virtual Destructor in C++ Article: https://www.geeksforgeeks.org/pure-virtual-destructor-c/

- ***Private Destructor:*** https://www.geeksforgeeks.org/private-destructor/
- ***Scope Of Destructors :*** https://www.geeksforgeeks.org/playing-with-destructors-in-c/

When an operator is overloaded with mutiple jobs, the mechanism of giving such special meanings to an operator it is known as operator overloading.

It provides a flexible option for the creation of new definitions for most of the C++ operations. It is a way to implement ***compile time polymorphism***.

Although the *semantics* of an operator can be extended, we cannot change its *syntax , grammatical* rules that govern its use such as the number of operands, precedence and associativity.

Also, when an operator is overloaded, its original meaning is not lost. For eg) operator + which can be overloaded to add two vectors can still be used to add two integers.

All operators in C++ { like **+** , **-** , > , < , >= , <= , % , / , * , ++ , -- , && , || , **[]** , **()** , **=** , += , -= , **>>** , **<<** , -> , ->* , new , delete , etc. } can be overloaded by adding keyword ***operator*** except few:

a. Class member access operators (**.** , **.***)
b. Scope Resolution operator (::)
c. "sizeof" operator
d. Conditional/ternary operator (?:)

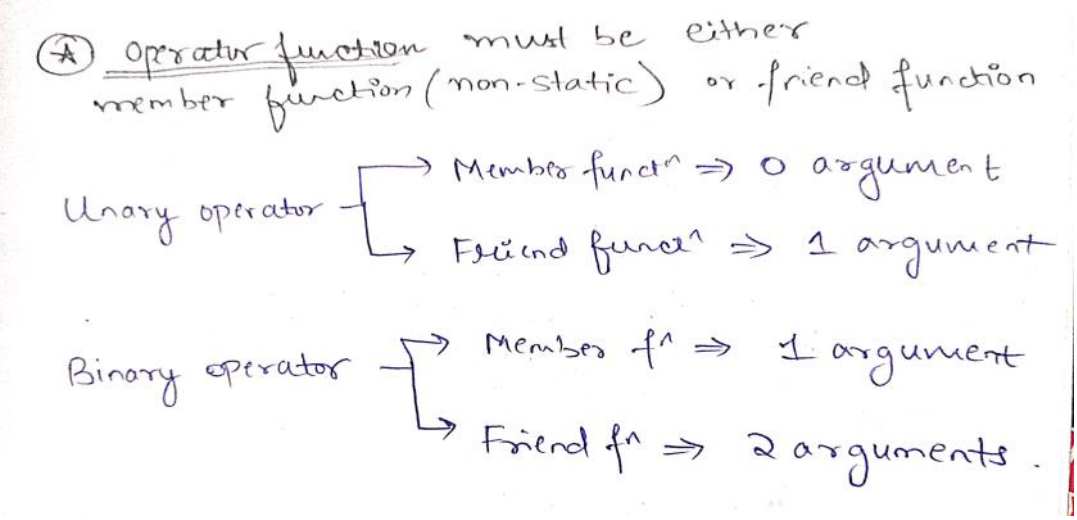
To define an additional task to an operator, we must specify what it means in relation to the class to which the operator is applied. This is done with the help of a special function known as ***operator function*** which describes the task.
It's general form is:

return-type class-name :: operator operator-symbol (argument list)
{ … Function Body… }

Here function name is combined keyword operator and operator-symbol which is to be overloaded.

Operator function must be either member function (non-static) or friend function.
Difference between them is that member function take one argument less than that in friend function as we need not pass the first operand on which overloaded operation need to be performed as object(first operand) used to invoke the operator function as member function is passed implicitly, but since friend function cannot access data members of class directly, first operand(object of class) also need to be passed (pass by reference otherwise modifications will not be showed up in object).
Thus for binary operator (which takes two operands objects), we need to pass only second operand object as argument to the member function while both objects as arguments to the friend function and for unary operator (which takes only one operand), we need no argument for

member function and one argument for friend function.

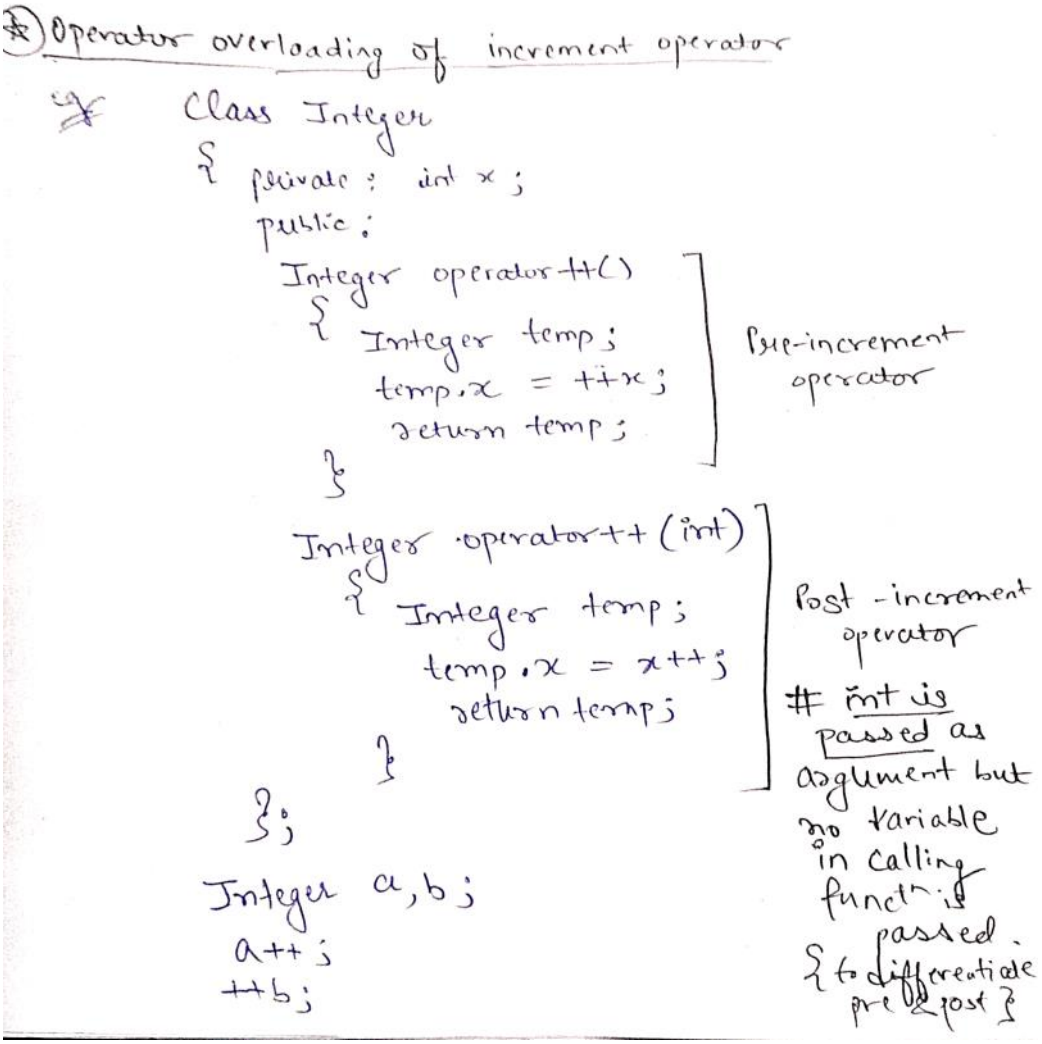


Note: If operator function is defined as member function, then it should be in public function as object(operand) need to call it just like a normal member function.

Overloading Unary Operators

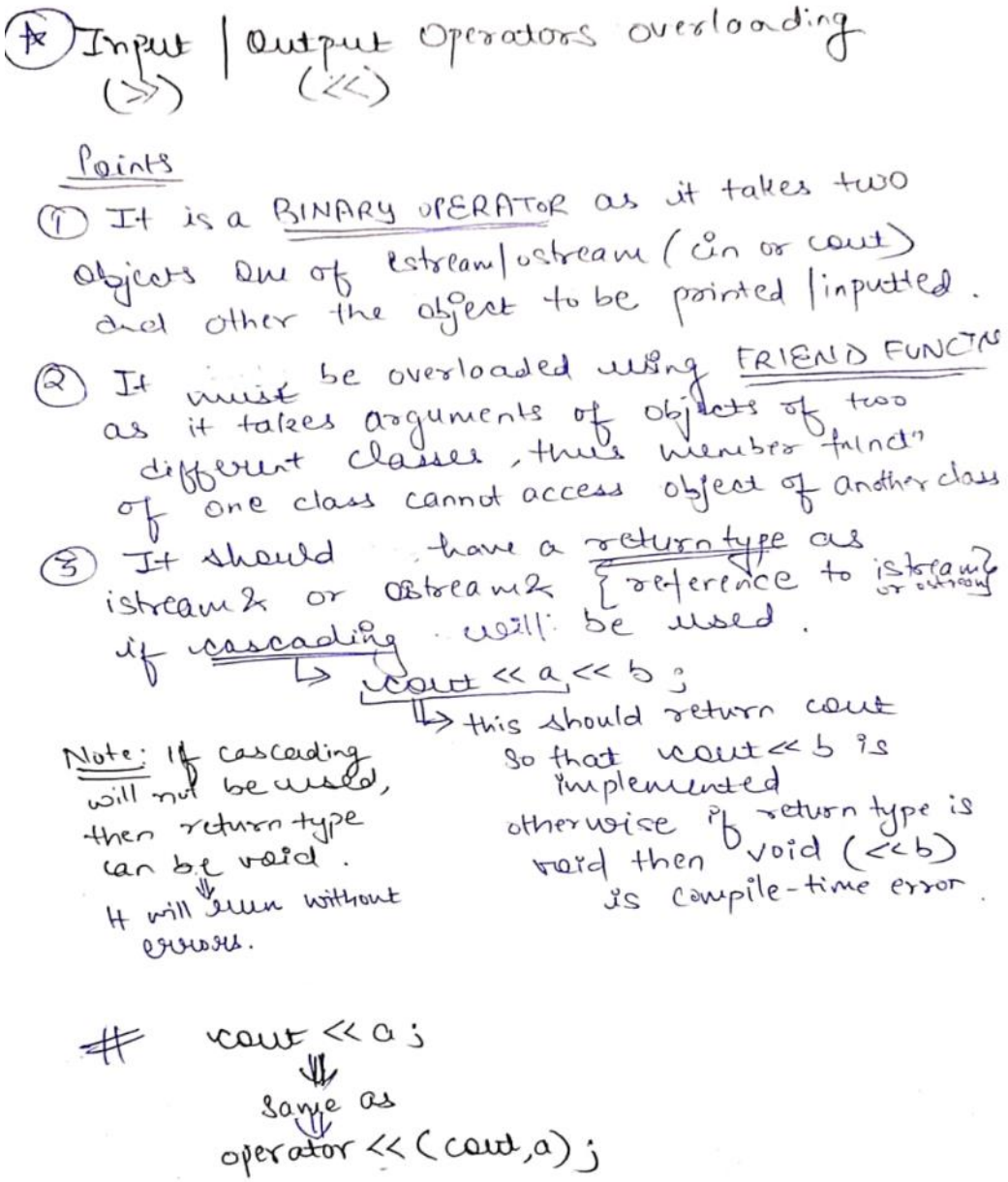
Some of the unary operators which can be overloaded are:

1. Unary minus (-)
2. Pre-increment and post-increment operator (++)



3. Pre-decrement and post-decrement operator (--)
4. Logical Not operator (!)

<u>Overloading Binary Operators</u>



Note: Overloading >> and << operator functions must be ***friend*** because it takes two arguments , one of type ostream/istream object and other of object of same class, now since cout/cin is the first object which is passed to ***operator<<(cout,obj);*** (or *cout<<obj;*) and it is an object of another class, thus it cannot be member function as operator function (as member function) takes first argument as object of same class only because it takes the first argument as calling object itself. If ***obj.operator<<(cout);*** had been the case then it was fine to declare it as member function because then first agument would had been (calling)object of same class.
Also if we will not declare it as friend function and leave it as a normal global function, then it cannot access private data members of the class. Thus, it needs to be friend function only.

Example of Operator Overloading)

```
class Distance {
public:
    // Member Object
```

```
    int feet, inch;
    // Overloading (+) binary operator to perform addition of two distance object using member function
    Distance operator+(Distance& d2) // Call by reference
    {
        // Create an object to return
        Distance d3;

        // Perform addition of feet and inches
        d3.feet = this->feet + d2.feet;
        d3.inch = this->inch + d2.inch;

        // Return the resulting object
        return d3;
    }
    // Or binary operator (+) overloading using friend function
    friend Distance operator+(Distance &d1, Distance& d2);

    // Overloading(-) unary operator to negate distance object using member function
    Distance operator-()
    {
        // Create an object to return
        Distance d;

        d.feet = -feet;
        d.inch = -inch;

        // Return the resulting object
        return d;
    }

    // Or unary operator (-) using friend function
    friend Distance operator-(Distance &s);

    // Insertion & Extraction Operator Overloading
    friend ostream &operator<<( ostream &output, const Distance &D );
    friend istream &operator>>( istream  &input, Distance &D );
};

ostream &operator<<( ostream &output, const Distance &D ) {
    output << "F : " << D.feet << " I : " << D.inches;
    return output;
  }

istream &operator>>( istream  &input, Distance &D ) {
    input >> D.feet >> D.inches;
    return input;
  }

Distance operator+(Distance &d1, Distance& d2)
{
    // Create an object to return
        Distance d3;

        // Perform addition of feet and inches
        d3.feet = d1.feet + d2.feet;
        d3.inch = d1.inch + d2.inch;

        // Return the resulting object
        return d3;
}

Distance operator-(Distance &s)
{
    s.feet = -s.feet;
    s.inch = -s.inch;

    // Return the resulting object
```

```
    return s;
}
```

Distance d1,d2;
Distance d3 = d1 + d2;
// This statement is same as d3 = d1.operator+(d2); , thus d1 is caller object and d2 is object passed as parameter.
when overloaded using member function
// This statement is same as d3 = operator+(d1,d2); when overloaded using friend function
Distance d4 = -d1;
// This statement is same as d4 = d1.operator-(); when overloaded using member function
but as d4 = operator-(d1); when overloaded using friend function
cout << d1 << d2 << d3 << d4;
// This statement is read as :
*(cascading can be done only if (cout << d1) itself return (cout) so that it runs as (cout << d2) for the next object i.e. insertion & extraction operator overloading function should have ostream/istream return type if **cascading** need to be done)*
operator<<(cout,d1);
operator<<(cout,d2);
operator<<(cout,d3);
operator<<(cout,d4);

Note: We can overload the operator functions also i.e. we can have implementation of **function overloading** of operator functions (used **for operator overloading**).
Example) Multiplying (overloading * operator) scalar(int) to a vector(class type) using function overloading (first function when scalar is first argument and other function where scalar is second argument)

```
class vector
{
    int v[size];
    public:
    friend vector operator*(int a,vector b);   // Overloaded Operator Function 1
    friend vector operator*(vector b,int a);  // Overloaded Operator Function 2
};
```

```
vector operator *(int a, vector b)
{
        vector c;

        for(int i=0; i < size; i++)
                c.v[i] = a * b.v[i];
        return c;
}

        vector operator *(vector b, int a)
{
        vector c;

        for(int i=0; i<size; i++)
                c.v[i] = b.v[i] * a;
        return c;
}
```

```
vector P, Q, R;
Q = 2 * R; // Calls function 1
P = Q * 2; // Calls function 2
```

Note: Similary like insertion & extraction operator overloading function 1 cannot be member function and need to be friend function as it takes *int* as first argument and not of calling object of same class *vector*. Although, function 2 can be member function as it takes first argument as calling object of same class, but to maintain consistency, both are declared as friend.

Rules for Operator Overloading

Although it looks simple to redefine the operators, there are certain restrictions and limitations in overloading them. Some of them are listed below:

- Only existing operators can be overloaded. New operators cannot be created.
- The overloaded operator must have at least one operand that is of user-defined type.
- We cannot change the basic meaning of an operators. Overloaded operators follow the Syntax rules of the original operators. They cannot be overridden.
- We cannot use friend functions to overload certain operators like assignment operator (=), function call operator (()), subscripting operator ([]) and class member access operator (->). However, member functions can be used to overload them.
- When using binary operators overloaded through a member function, the left hand operand must be an object of relevant class(of calling object)
- Binary arithmetic operators such as +, -, *, / must explicitly return a value. They must not attempt to change their own arguments.

Advantages of Type Conversion

- This is done to take advantage of certain features of type hierarchies or type representations.
- It helps to compute expressions containing variables of different data types.

C++ allows us to convert data of one type to that of another. This is known as **Type Conversion**.

There are two types of type conversion in C++.

1. **Implicit Conversion**

   The type conversion that is done automatically done by the compiler when a value is copied (using assignment operator) to a compatible type is known as implicit type conversion. This type of conversion is also known as ***automatic conversion***. Implicit conversions do not require any operator.

   Standard conversions affect primitive data types, and allow conversions such as between numerical types (short to int, int to float, double to int...), to or from bool, and some pointer conversions. Thus they are automatic as long as the data types involved are built-in types.

   Example 1: Conversion From int to double

   ```
   int i = 9;
   double d = i;
   cout << i << d;  // => prints 9 9
   ```

   Example 2: Conversion From double to int

   ```
   double d = 9.9;
   int i = d;
   cout << i << d;  // => prints 9 9.9
   ```

   Note: Implicit conversion in mixed expressions(where left and right operands are of different type) is applied only after completing all ***integral widening conversions***.

   For eg) Whenever a *char* or *short int* appears in an expression, it is converted to an int and this is known as integral widening conversions. This is done to convert smaller datatype to wider type to avoid data loss. This is done using ***waterfall model*** of type conversions.

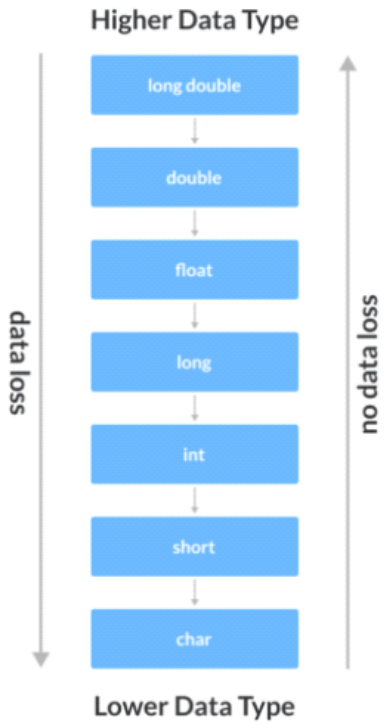
Fig. 3.3 ⇔ *Water-fall model of type conversion*

Table 3.4 *Results of Mixed-mode Operations*

RHO / LHO	*char*	*short*	*int*	*long*	*float*	*double*	*long double*
char	int	int	int	long	float	double	long double
short	int	int	int	long	float	double	long double
int	int	int	int	long	float	double	long double
long	long	long	long	long	float	double	long double
float	float	float	float	float	float	double	long double
double	double	double	double	double	double	double	long double
long double	long double	long double	long double	long double	long double	long double	long double

RHO – Right-hand operand
LHO – Left-hand operand

Note: Since *int* cannot have a decimal part, the digits after the decimal point are truncated in Eg2. Thus, conversion from one data type to another is prone to **data loss**. This happens when data of a larger type is converted to data of a smaller type. Thus compiler gives **warnings** in such case which is known as ***narrowing conversion***.



Note: Implicit conversions also include ***constructor or operator conversions***, which affect classes that include specific constructors or operator functions to perform conversions. For example:

```
class A {};
class B { public: B (A a) {} };
A a;
B b = a;
```

Here, an implicit conversion happened between objects of class A and class B, because B has a constructor that takes an object of class A as parameter. Therefore implicit conversions from A to B are allowed.

2. **Explicit Conversion** (also known as **Type Casting**)

   When the user manually changes data from one type to another, this is known as **explicit conversion**. This type of conversion is also known as **type casting**.

   There are three major ways in which we can use explicit conversion in C++. They are:

   a. C-style type casting (also known as **cast notation**)
      The syntax for this style is: ***(data_type)expression;***
      For eg) double d = (double)i; or int i = (int)d;

   b. Old C++ styple type casting (also known as **Function notation**)
      The syntax for this style is: ***data_type(expression);***
      For eg) double d = double(i); or int i = int(d);

   Note: p = int*(q) is illegal, it should be p = (int*)q;. Hence b. should be used only if the type is

an identifier.
Alternatively we can use ***typedef*** to create identifier of required type and use it in b.
Eg) *typedef int* int_ptr;*
p = int_ptr(q);

Note: Traditional explicit type-casting (a. & b.) allows to convert any pointer into any other pointer type, independently of the types they point to. This will not give syntatical errors but can produce runtime errors or unexpected behaviors.

Example of condition where run-time error will occur:

```
#include <iostream>
using namespace std;
class CDummy {
   float i,j;
};
class CAddition {
        int x,y;
  public:
        CAddition (int a, int b) { x=a; y=b; }
        int result() { return x+y;}
};
int main () {
  CDummy d;
  CAddition * padd;
  padd = (CAddition*) &d;
  cout << padd->result();
  return 0;
}
```

c. New C++ style type casting using **Type conversion operators**
C++11 has introduced 4 operators for type conversions:

i. static_cast
static_cast can perform conversions between pointers to related classes, not only from the derived class to its base, but also from a base class to its derived. This ensures that at least the classes are compatible if the proper object is converted, but no safety check is performed during runtime to check if the object being converted is in fact a full object of the destination type. Therefore, it is up to the programmer to ensure that the conversion is safe. On the other side, the overhead of the type-safety checks of dynamic_cast is avoided.
static_cast can also be used to perform any other non-pointer conversion that could also be performed implicitly, like for example standard conversion between fundamental types.
https://www.geeksforgeeks.org/static_cast-in-c-type-casting-operators/

static_cast In C++ | What Is static_cast In C++

ii. dynamic_cast

dynamic_cast can be used only with pointers and references to objects. Its purpose is to ensure that the result of the type conversion is a valid complete object of the requested class.

Therefore, dynamic_cast is always successful when we cast a class to one of its base classes. Base-to-derived conversions are not allowed with dynamic_cast unless the base class is polymorphic (has atleast one virtual function). When a class is polymorphic, dynamic_cast performs a special checking during runtime to ensure that the expression yields a valid complete object of the requested class.

dynamic_cast In C++ | How To Use dynamic_cast In C++?

Note: dynamic_cast requires the Run-Time Type Information (RTTI) to keep track of dynamic types. Refer: https://www.geeksforgeeks.org/g-fact-33/

iii. const_cast

This type of casting manipulates the constness of an object, either to be set or to be removed. For example, in order to pass a const argument to a function that expects a non-constant parameter.
https://www.geeksforgeeks.org/const_cast-in-c-type-casting-operators/

const_cast In C++ | Where To Use const_cast In C++?

iv. reinterpret_cast

reinterpret_cast converts any pointer type to any other pointer type, even of unrelated classes. The operation result is a simple binary copy of the value from one pointer to the other. All pointer conversions are allowed: neither the content pointed nor the pointer type itself is checked.

It can also cast pointers to or from integer types. The format in which this integer value represents a pointer is platform-specific. The only guarantee is that a pointer cast to an integer type large enough to fully contain it, is granted to be able to be cast back to a valid pointer.

The conversions that can be performed by reinterpret_cast but not by static_cast are low-level operations, whose interpretation results in code which is generally system-specific, and thus non-portable.

https://www.geeksforgeeks.org/reinterpret_cast-in-c-type-casting-operators/

reinterpret_cast In C++ | Where To Use reinterpret_cast In C++?

Type Conversion for User Defined Datatypes

Since user defined data types are designed by programmer to suit his requirements, the compiler does not support automatic type conversions for such data types.

Hence, we **must** design the conversion routines by ourselves is such operations are required, otherwise compiler will give compile-time errors if not handled.

These types of situation might arise in data conversion b/w uncompatible types:

1. ***Conversion from Primitive to Non-Primitive(Class) Type Conversion***

Eg) class Obj;
int a;
Obj = a;

Parameterized Constructor in Class is required which will be having one argument of the datatype from which conversion needs to take place (or type conversion while passing the argument to constructor will take). Constructor performs a *defacto type conversion* from the argument's type to the constructor's class type.

```
Eg) class Time
    {
        int Hrs, Mins;
        public:
        Time(int t)
        {
            Hrs = t/60;
            Mins = t%60;
        }
    };
    Time t1;
    int duration = 85;
    T1 = duration;
```

Lecture 24 Type Conversion Primitive to class type in C++ Hindi

2. ***Conversion from Non-Primitive(Class) to Primitive Type Conversion***

C++ allows us to define an ***overloaded casting operator*** that could be used to convert a class type to a primitive type. The overloaded casting operator is also known as **conversion function**. Syntax of Casting Operator:

```
operator typename()
{
    // Function Statements for Type Conversion.
}
```

This conversion function converts a class type data to *typename* (primitive datatype). This function have keyword operator (for implementing operator overloading).
It ***must not have any return type and no arguments list*** and it ***must be a class member***.
Since it is a member function, it is invoked by the object and therefore, the values used for conversion inside the function belong to the object that invoked the function. This means that the function does not need an argument.

Eg) Consider a class *Vector* whose we need to find the scalar magnitude (square roots of the sum of squares of components). We need to initialize a primitive datatype (double) with this magnitude.

```
Vector :: operator double()
{
    double Sum = 0;
    for(int i=0; i<size; i++)
        Sum = Sum + v[i] * v[i];
    return sqrt(Sum);
}
Vector v1;
double length = v1;
// or we can write double length = double(v1) showing explicit conversion.
```

Lecture 25 Type Conversion Class type to primitive type in C++ Hindi

3. ***From Non-Primitive(One Class) to Non-Primitive(another different class) Type Conversion***

```
Eg) ClassX  A;
    ClassY  B;
    A = B;
```

Since conversion takes place from classY to classX, classY is known as *source class* and classX is

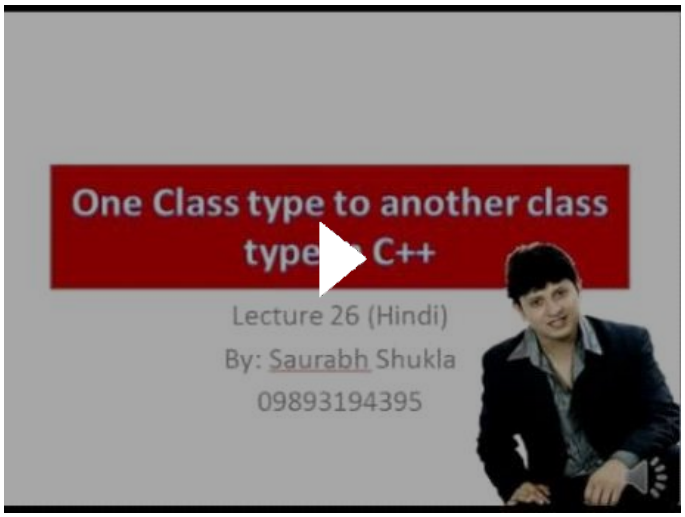
known as *destination class*.
Such conversions b/w objects of different classes can be carried out by **either a parameterized constructor of destination class(X) or a conversion function (overloaded conversion operator) of source class(Y)**. Both will be treated the same way by the compiler.

If conversion function of source class(Y) is used, then *typename* will be a user-defined type i.e. of destination class(X). But if constructor of destination class(X) is used, we must be able to access the data members of the object sent (of source class Y) as an argument. Since data members of the source class are private, we must use special access functions (getters and setters) in source class(Y) to facilitate its data flow to the destination class(X).

Note: It is important that we **do not use both** the constructor in source class and casting operator in destination class for the same type conversion, as this will **cause an ambiguity** that how type conversion should be performed.

Note: If both objects are of same class type, then type conversion is not required, as it a simple assignment operator which will do a shallow or deep copy based on whether assignment operator is overloaded(and pointers handled accordingly) or not.

Lecture 26 Type Conversion one class type to another class type in C++ Hindi



Virtual Function

Function (or Method) Overriding

Lecture 17 Method Overriding in C++ Part 1 Hindi

When the base class and derived class have member functions with exactly the **same name**, **same return-type**, and **same arguments list**, then it is said to be function overriding.

Function overriding is a feature that allows us to have a same function in child class which is already present in the parent class. A child class inherits the data members and member functions of parent class, but when you want to override a functionality in the child class then you can use function overriding. It is like creating a new version of an old function, in the child class.

Note: In function overriding, the function in parent class is called the ***overridden function*** and function in child class is called ***overriding function***.

Example)

```
class A {
  public:
  void display() {
    cout<<"Base class";
  }
};
class B:public A {
  public:
  void display() {
    cout<<"Derived Class";
  }
};
int main() {
  B obj;
  obj.display();
  return 0;
}
```

It will produce the following output

```
Derived Class
```

Function Overloading vs Function Overriding

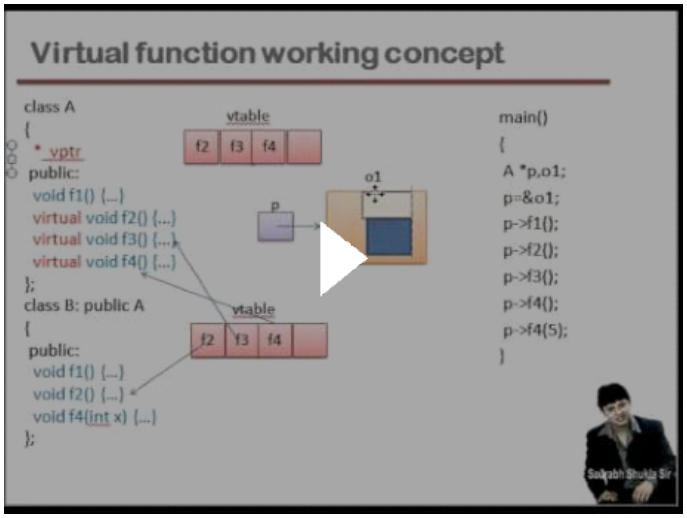
- **Definition**: Function overloading provides multiple definitions of the function by changing signature i.e changing number of parameters, change datatype of parameters, return type doesn't play any role. Function overriding is the redefinition of base class function in its derived class with same signature i.e return type and parameters.
- **Inheritance**: Overriding of functions occurs when one class is inherited from another class. Overloading can occur without inheritance.
- **Function Signature**: Overloaded functions must differ in function signature i.e. either number of parameters or type of parameters should differ. In overriding, function signatures must be same.
- **Scope of functions**: Overridden functions are in different scopes, whereas overloaded functions are in same scope. Function overloading can be done in **any class** , base or derived, **or even for global** functions (non-member functions) while function overriding can only be done in **derived class**.
- **Behavior of functions**: Overriding is needed when derived class function has to do some added or different job than the base class function. Overloading is used to have same name functions which behave differently depending upon parameters passed to them.

Q) If we have a function in base class and a function with same name in derived class, can the base class function be called from derived class object?
R) Overloading does not work with Inheritance . Read => https://www.geeksforgeeks.org/does-overloading-work-with-inheritance/

Virtual Functions

Lecture 18 Virtual Function in C++ Part 1 Hindi

Lecture 18 Virtual Function in C++ Part 2 Hindi



Lecture 19 Abstract Class in C++ Part 1 Hindi

Polymorphism refers to the by which objects belonging to different classes are able to respond to the same message but in different forms. An essential requirement of polymorphism is therefore the ability to refer to objects without any regard to their classes. This necessitates the use of a single pointer variable to refer to the obects of different classes. Hence, we the use pointer to base class to refer to all the derived objects.

But, a base pointer, even though made to contain the address of a derived class, always executes the function in the base class due to early binding in ***function overriding***. The compiler simply ignores the contents of the pointer and choses the member function that matches the type of the pointer. Thus to **achieve run-time polymorphism**, we require the ***virtual functions***.

When we use the same function name in both the base and derived classes, the function in base class is declared as virtual using the keyword *virtual* preceding its normal declaration. When a function is made virtual, C++ determines which function to use at run-time based on the **type of object pointed to by the base pointer, rather than the type of the pointer**. Thus, by making the base pointer to point to different(derived class) objects, we can execute different versions of the virtual function (overrided version in derived class).

In short, virtual functions ensure that the correct function is called for an object, regardless of the type of reference (or pointer) used for function call, by resolving the function call at run-time.

Late binding(Runtime) is done in accordance with the content of pointer (i.e. location pointed to by pointer) and Early binding(Compile time) is done according to the type of pointer.

Note: If we have created a virtual function in the base class and it is being overridden in the derived class then we don't need virtual keyword in the derived class, functions are automatically considered as virtual functions in the derived class.

VIRTUAL FUNCTIONS

```
#include <iostream>

using namespace std;

class Base
{
  public:
        void display() {cout << "\n Display base ";}
        virtual void show() {cout << "\n show base";}
};
class Derived : public Base
{
  public:
        void display() {cout << "\n Display derived";}
        void show() {cout << "\n show derived";}
};

int main()
{
        Base B;
        Derived D;
        Base *bptr;

        cout << "\n bptr points to Base \n";
        bptr = &B;
        bptr -> display();   // calls Base version
        bptr -> show();      // calls Base version

        cout << "\n\n bptr points to Derived\n";
        bptr = &D;
        bptr -> display();   // calls Base version
        bptr -> show();      // calls Derived version

        return 0;
}
```

PROGRAM 9.12

The output of Program 9.12 would be:

```
bptr points to Base

Display base
Show base

bptr points to Derived

Display base
Show derived
```

note

When **bptr** is made to point to the object **D**, the statement

```
bptr -> display();
```

calls only the function associated with the **Base** (i.e. **Base :: display( )**), whereas the statement

```
bptr -> show();
```

calls the **Derived** version of **show()**. This is because the function **display()** has not been made **virtual** in the **Base** class.

RUNTIME POLYMORPHISM

```
#include <iostream>
#include <cstring>

using namespace std;

class media
{
  protected:
        char  title[50];
        float price;
  public:
        media(char *s, float a)
        {
               strcpy(title, s);
               price = a;
        }
        virtual void display() { }   // empty virtual function
};

class book: public media
{
        int pages;
  public:
        book(char *s, float a, int p):media(s,a)
        {
               pages = p;
        }
        void display();
};
```

```
class tape :public media
{
        float time;
  public:
        tape(char * s, float a, float t):media(s, a)
        {
               time = t;
        }
        void display();
};

void book :: display()
{
        cout << "\n Title: " << title;
        cout << "\n Pages: " << pages;
        cout << "\n Price: " << price;
}
void tape :: display()
{
        cout << "\n Title: " << title;
        cout << "\n play time: " << time << "mins";
        cout << "\n price: " << price;
}

int main()
{
        char * title = new char[30];
        float price, time;
        int pages;

        // Book details
        cout << "\n ENTER BOOK DETAILS\n";
        cout << " Title:"; cin >> title;
        cout << " Price: "; cin >> price;
        cout << " Pages: "; cin >> pages;

        book book1(title, price, pages);

        // Tape details
        cout << "\n ENTER TAPE DETAILS\n";
        cout << " Title: "; cin >> title;
        cout << " Price: "; cin >> price;
        cout << " Play time (mins): "; cin >> time;
```

```
    tape tape1(title, price, time);

    media* list[2];
    list[0] = &book1;
    list[1] = &tape1;

    cout << "\n MEDIA DETAILS";

    cout << "\n ......BOOK......";
    list[0] -> display(); // display book details

    cout << "\n ......TAPE......";
    list[1] -> display(); // display tape details

    result 0;
}
```

PROGRAM 9.13

The output of Program 9.13 would be:

```
ENTER BOOK DETAILS
Title:Programming_in_ANSI_C
Price: 88
Pages: 400

ENTER TAPE DETAILS
Title: Computing_Concepts
Price: 90
Play time (mins): 55

MEDIA DETAILS
......BOOK......
Title:Programming_in_ANSI_C
Pages: 400
Price: 88

.....TAPE......
Title: Computing_Concepts
Play time: 55mins
Price: 90
```

Rules for Virtual Functions

- Virtual functions **cannot be *static*** and also **cannot be a *friend*** function of another class, i.e. they **need to be non-static member functions** of the class
- Virtual functions should be **accessed using pointer or reference of base class type** to achieve run time polymorphism.
  (If an object of base class type is made {not pointer to object} and member function is called using dot operator, it will call base class member function only and not the overriden derived class member function as object is of base class type and no polymorphism can be achieved)
- The **prototype** of virtual functions should be **same in base as well as derived class** (i.e. function should be overriden not overloaded or hidden otherwise mechanism of virtual functions will be ignored.)
- They are always defined in base class and overridden in derived class. It is not mandatory for derived class to override (or re-define the virtual function), in that case base class version of function is used.
- A class **may have virtual destructo**r but it **cannot have a virtual constructor**.
- While a base pointer can point to any type of the derived object, the reverse is not true i.e. we **cannot use a pointer to a derived class to access an object of the base type**.
- When a base pointer points to a derived class, incrementing or decrementing it will not make it to point to the next object of the derived class. It is incremented or decremented only relative to its base type. Therefore, we **should not use this method to move the pointer to the next object**.
- If a **virtual function** is defined in the base class, it **need not be necessarily redefined (overrided)** in the derived class. In such cases, calls will invoke the base function.

Working of virtual functions(concept of VTABLE and VPTR)

If a class contains a virtual function then compiler itself does two things:

- If object of that class is created then a virtual pointer(*vptr*) is inserted as a data member of the class to point to *vtable* of that class. For each new object created, a new virtual pointer is inserted as a data member of that class.
- Irrespective of object is created or not, a static array of function pointer called *vtable* where each cell contains the address of each virtual function contained in that class.

Compiler adds additional code at two places to maintain and use *vptr*.
1) Code in every constructor. This code sets the vptr of the object being created. This code sets vptr to point to the vtable of the class.
2) Code with polymorphic function call. Wherever a polymorphic call is made, the compiler inserts code to first look for vptr using base class pointer or reference. Once vptr is fetched, vtable of derived class can be accessed. Using vtable, address of derived derived class function show() is accessed and called.

Private Virtual Functions in C++ => https://www.geeksforgeeks.org/can-virtual-functions-be-private-in-c/

Pure Virtual Functions
A pure virtual function is a virtual function in C++ for which we need not to write any function definition and only we have to declare it. It is declared by assigning 0 in the declaration.
It is normal practice to declare a function virtual inside the base class and redefine it in the derived classes. The function inside the base class is seldom used for performing any task. It only serves as a placeholder. Such functions are called 'do-nothing' functions.
A pure virtual function is a function declared in a base class that has no definiton relative to the base class. In such cases, the compiler requires each derived class to either define the function or redeclare it as a pure virtual function.

Example)
```
class B {
  public:
    virtual void s() = 0; // Pure Virtual Function
};

class D:public B {
  public:
    void s() {
      cout << "Virtual Function in Derived class\n";
    }
};

int main() {
  B *b;
  D dobj;
  b = &dobj;
  b->s();
}
```

Output
```
Virtual Function in Derived class
```

Abstract Class
An **abstract class** is a class in C++ which have at least one pure virtual function. It is not used to create objects.
An abstract Class is only designed to act as a base class to be inherited by other classes.

- Abstract class can have normal functions, constructors and variables along with a pure virtual function.
- Abstract class cannot be instantiated, but pointers and references of Abstract class type can be created.
- Abstract classes are mainly used for Upcasting, so that its derived classes can use its interface.
- If an Abstract Class has derived class, they must implement all pure virtual functions, or else they will become Abstract too i.e. If we do not override the pure virtual function in derived class, then derived class also becomes abstract class.
- We **can't create object of abstract class** as we reserve a slot for a pure virtual function in Vtable, but we don't put any address, so Vtable will remain incomplete. In short, ***Instantiation of abstract class is not possible***.
- We can have pointers and references of abstract class type.

Note: Unlike Java, there is no 'abstract' keyword in C++ and is implemented only using class having atleast one pure virtual function.

Note: Classes that can be used to instantiate objects are called **concrete classes**. All classes that are not abstract are concrete.

Note: Interface vs Abstract Classes

- An interface does not have implementation of any of its methods, it can be considered as a collection of method declarations.
- In C++, an interface can be simulated by making all methods as pure virtual.
- An interface describes the behavior or capabilities of a C++ class without committing to a particular implementation of that class.
- *When to use Interface vs When to use Abstract classes?*
  If you need multiple inheritance and a clear blueprint than has only the design and not the implementation; you go for interface. If you don't need multiple inheritance but you need a mix of design plan and pre-implementation then abstract class is your choice.

Note: Java has 'interface' keyword for implementing interface classes but interface can only be implemented using class having all functions as pure virtual.

Difference between Interface and Absract Class (In Java)

Abstract class	Interface
1) Abstract class can **have abstract and non-abstract** methods.	Interface can have **only abstract** methods. Since Java 8, it can have **default and static methods** also.
2) Abstract class **doesn't support multiple inheritance**.	Interface **supports multiple inheritance**.
3) Abstract class **can have final, non-final, static and non-static variables**.	Interface has **only static and final variables**.
4) Abstract class **can provide the implementation of interface**.	Interface **can't provide the implementation of abstract class**.
5) The **abstract keyword** is used to declare abstract class.	The **interface keyword** is used to declare interface.
6) An **abstract class** can extend another Java class and implement multiple Java interfaces.	An **interface** can extend another Java interface only.
7) An **abstract class** can be extended using keyword "extends".	An **interface** can be implemented using keyword "implements".
8) A Java **abstract class** can have class members like private, protected, etc.	Members of a Java interface are public by default.

9)**Example:**
public abstract class Shape{
public abstract void draw();
}
	Example:
	public interface Drawable{
	void draw();

Virtual Function vs Pure Virtual Function

Similarities between virtual function and pure virtual function

- These are the concepts of Run-time polymorphism.
- Prototype i.e. Declaration of both the functions remains the same throughout the program.
- These functions can't be global or static.

Difference between virtual function and pure virtual function

VIRTUAL FUNCTION	PURE VIRTUAL FUNCTION
A virtual function is a member function of base class which can be redefined by derived class.	A pure virtual function is a member function of base class whose only declaration is provided in base class and should be defined in derived class otherwise derived class also becomes abstract.
Classes having virtual functions are not abstract.	Base class containing pure virtual function becomes abstract.
Syntax: `virtual<func_type><func_name>() {  // code }`	Syntax: `virtual<func_type><func_name>() = 0;`
Definition is given in base class.	No definition is given in base class.
Base class having virtual function can be instantiated i.e. its object can be made.	Base class having pure virtual function becomes abstract i.e. it cannot be instantiated.
If derived class do not redefine virtual function of base class, then it does not affect compilation.	If derived class do not redefine virtual function of base class, then no compilation error but derived class also becomes abstract just like the base class.
All derived class may or may not redefine virtual function of base class.	All derived class must redefine pure virtual function of base class otherwise derived class also becomes abstract just like base class.

Eg) *int x = 5;*

*int *ptr = &x; //* ***int* ptr or int * ptr or int *ptr*** *all are valid*

=> *x = *(&x) = *ptr = 5;*

ptr = &x = 2048;

Addressof Operator (&)

It is an unary operator whose operand must be a variable. It is used by pointers to refer to a memory address/location.

This operator is also known as ***referencing operator***.

ptr is a pointer which is pointing to the memory address of variable *x* using *addressof operator(&)*.

Note: ptr itself takes 4 bytes of memory (equal to *sizeof(int),* all pointers irrespective of type takes 4 bytes only) whose memory address is different from memory address of x.

Deferencing Operator (*)

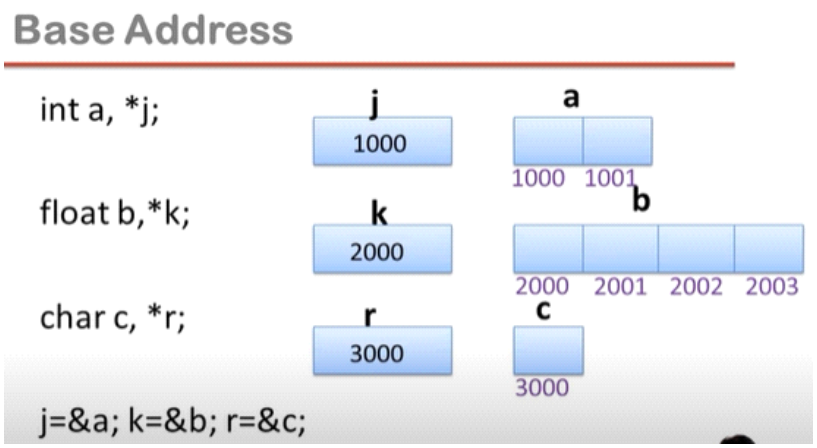
It is an unary operator and takes memory address as operand. It returns the content/value at the address location of operand. It is also known as ***indirection operator***.

*ptr { or *(&x) } will deference address of x and gives the value at memory address 2048 which is 5.

Q) Why pointer cannot be generic or without any datatype i.e. why pointers must have a datatype which is same as the datatype they are pointing to, like why int pointer should point to integer data only and so on?

R) Since we require **to implement *pointer deferencing* and *pointer arithmetic***, we **need the *sizeof the datatype*** which pointer is pointing to. For eg) float will take 4 bytes and if we make datatype of pointer pointing to float data other than float like char which takes 1 byte, then on dereferencing it, it will read only 1 byte out of 4 bytes and don't give the value of float data. Also doing pointer arithmetic, p++ will make jump in 1byte only but getting next float data require a jump of 4 bytes. Also, if we don't give any datatype, then also compiler will not know how many bytes to take into consideration for deferencing and arithmetic.

There is generic pointer known as *void pointer* in C++, but pointer arithmetic is not possible with void pointer and dereferencing is also not possible. Use of void (generic) pointer is discussed later.



Pointer Expressions and Pointer Arithmetic

A limited set of arithmetic operations can be performed on pointers. A pointer may be:

- incremented (++)
  { increments by size of datatype to which pointer is pointing (type of pointer).
  For eg) for integer array ptr++ will point to next array index address location i.e. increment size of int to previous location. }
- decremented (--)
- an integer may be added to a pointer (+ or +=)
- an integer may be subtracted from a pointer (– or -=)

Pointer arithmetic is meaningless unless performed on an array/string.

Note :

Pointers contain addresses. Adding two addresses makes no sense, because there is no idea what it would point to.

Subtracting two addresses lets you compute the offset/distance between these two address locations.

However, difference of two address gives (address of one - address of other)/sizeof(datatype of pointers).

REFERENCES

When a variable is declared as reference, it becomes an alternative name for an existing variable. A variable can be declared as reference by putting '&' in the declaration.

Applications :

1. **Modify the passed parameters in a function** : If a function receives a reference to a variable, it can modify the value of the variable. For example, in the following program variables are swapped using references.

```
#include<iostream>
using namespace std;

void swap (int& first, int& second)
{
    int temp = first;
    first = second;
    second = temp;
}

int main()
{
    int a = 2, b = 3;
    swap( a, b );
    cout << a << " " << b;
    return 0;
}
```

Output:

```
3 2
```

2. **Avoiding copy of large structures** : Imagine a function that has to receive a large object. If we pass it without reference, a new copy of it is created which causes wastage of CPU time and memory. We can use references to avoid this.

```
struct Student {
   string name;
   string address;
   int rollNo;
}

// If we remove & in below function, a new
// copy of the student object is created.
// We use const to avoid accidental updates
// in the function as the purpose of the function
// is to print s only.
void print(const Student &s)
{
    cout << s.name << "  " << s.address << "  " << s.rollNo;
}
```

3. **In For Each Loops to modify all objects** : We can use references in for each loops to modify all elements

```
#include <bits/stdc++.h>
using namespace std;

int main()
{
    vector<int> vect{ 10, 20, 30, 40 };

    // We can modify elements if we
    // use reference
    for (int &x : vect)
        x = x + 5;

    // Printing elements
    for (int x : vect)
       cout << x << " ";

    return 0;
}
```

4. **In For Each Loops to avoid copy of objects** : We can use references in for each loops to avoid copy of individual objects when objects are large.

```
#include <bits/stdc++.h>
using namespace std;

int main()
{
    vector<string> vect{"geeksforgeeks practice",
                        "geeksforgeeks write",
                        "geeksforgeeks ide"};

    // We avoid copy of the whole string
    // object by using reference.
    for (const auto &x : vect)
       cout << x << endl;

    return 0;
}
```

References vs Pointers

Similarities

Both references and pointers can be used to change local variables of one function inside another function. Both of them can also be used to save copying of big objects when passed as arguments to functions or returned from functions, to get efficiency gain.

Differences

- Void : A pointer can be declared as void but a reference can never be void.
- Initialization: A reference must be initialized when declared i.e declared and intiliased in the same line. There is no such restriction with pointers.
- Reassignment: A pointer can be re-assigned. This property is useful for implementation of data structures like linked list, tree, etc. But reference cannot be reassigned to another object after it is initialized to an address.
- Memory Address: A pointer has its own memory address and size on the stack whereas a reference shares the same

memory address (with the original variable) but also takes up some space on the stack.

- NULL Value : References cannot be NULL. Pointers are often made NULL to indicate that they are not pointing to any valid thing.
- Indirection: You can have pointers to pointers offering extra levels of indirection. Whereas references only offer one level of indirection.
- Arithmetic operations: Various arithmetic operations can be performed on pointers whereas there is no such thing called Reference Arithmetic.

Due to the above limitations, references in C++ cannot be used for implementing data structures like Linked List, Tree, etc. and when NULL is to be passed as pointer like in arrays.

References are safer and easier to use:

1) Safer: Since references must be initialized, wild references like wild pointers are unlikely to exist. It is still possible to have references that don't refer to a valid location (See questions 5 and 6 in the below exercise)

2) Easier to use: References don't need dereferencing operator to access the value. They can be used like normal variables. '&' operator is needed only at the time of declaration. Also, members of an object reference can be accessed with dot operator ('.'), unlike pointers where arrow operator (->) is needed to access members.

Together with the above reasons, there are few places like copy constructor argument where pointer cannot be used. Reference must be used pass the argument in copy constructor. Similarly references must be used for overloading some operators like ++.

Note: Function having return type as reference can be taken as L Value.
For eg)

```
int& increment(int &a)
{ return &a; }
int main()
{
    int a = 5;
    int b = 6;
    increment(a)  = b; // Function as L value
}
```

Reference Variables

Lecture 4 Reference Variables in C++ Part 1 hindi

A reference variable provides an *alias* (alternative name) for a previously defined variable.
Syntax: ***data_type & reference_name = variable_name;***

Unlike declaration of pointer (using *), it is declared using & before reference_name. But it does not mean that *reference_name* will hold the address of *variable_name* as **& is not '*addressof*' operator here** it just means reference to data_type. Unlike pointer, they cannot be deferenced using * operator since they do not hold memory address but value only.

Reference variables **do not require extra memory** space since they refer to the same variable already in memory just with a different name. If we print *&reference_variable,* we will get *&variable_name,* only since they both refer to same memory with different names.

A reference variable **must be initialized at the time of declaration** i.e. dynamic initialization of reference variable is not possible. Thus they must be initialized with a valid variable (which is already declared earlier) in the same line in which they are declared. Though variable they refer to can just be declared (before reference variable declaration) and not initialized. Also they cannot be initialized with even NULL or nullptr.

Once initialized(during declaration itself) with a *variable_name,* they cannot be updated to refer to a different variable i.e. they can refer to a single variable only, **reassignment of reference variable is not possible**.
Eg) int x = 5, y = 10;
    int &z = x;
    z = y; // This will not assign or update z to y but just change value of x from 5 to 10(y).

Reference variable is an ***internal pointer***, i.e. it is implemented using pointers internally, but it cannot refer to the address of variable (using &) and also it cannot be dereferenced (using *) since it

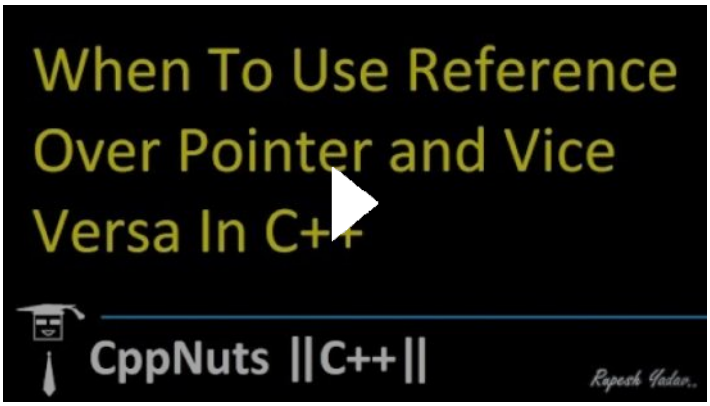
is not an pointer and just another name for the variable.
Also, since it is not pointer, pointer arithmetic cannot be applied i.e. addition or subtraction of *reference_name* will cause addition or subtraction (operation) to the *variable_name* directly and not to the address locations.

Note: Following reference is also allowed:

```
int x;
int *p = &x;
int &m = *p; // Reference to Deferenced(*) Variable x
```

Note: References can not only be created for built-in datatypes but also for user-defined datatypes like structures and classes.

When To Use Reference Over Pointer and Vice Versa In C++



Note: References should be used over pointers in following cases:

1. Call/Pass by Reference
2. To avoid ***object slicing***
3. To call overrided function(in derived class) instead of base class function by passing by reference instead of pass by value i.e. to achieve *runtime polymorphism*
4. To modify local variable of caller function using functions.
5. Where references **must** have to be used like in copy constructor

Note: Object Slicing: https://www.youtube.com/watch?v=ezgq_fShJw4

Pointers should be used over references in following cases:

1. When variable need to be initialized with NULL or nullptr
2. When reassignment/updation of variable is required. Pointer can point to different memory address but reference once initiliazed cannot be assiged to refer to another variable.

Function Call Types

The parameters passed to function are called ***actual parameters*** whereas the parameters received by function are called ***dummy*** *or* ***formal parameters***.

Functions can be invoked in 3 ways:

1. **Call by value**
   The call by value method of passing arguments to a function copies the value of actual arguments into the formal parameter of the function and the two types of parameters are stored in different memory locations. In this case, changes made to the parameter inside the function have no effect on the actual arguments(arguments of caller).

   By default, C++ uses call by value to pass arguments. In general, this means that code within a function cannot alter the arguments used to call the function.

   Example)

```
void swap(int x, int y)
{
        int temp;

        temp = x;
        x = y;
        y = temp;
        return;
}
int main ()
{
  int a = 100;
  int b = 200;
  cout << "Before swap, value of a :" << a << endl;
  cout << "Before swap, value of b :" << b << endl;
  swap(a, b);
  cout << "After swap, value of a :" << a << endl;
  cout << "After swap, value of b :" << b << endl;
  return 0;
}
```

OUTPUT:

```
Before swap, value of a :100
Before swap, value of b :200
After swap, value of a :100
After swap, value of b :200
```

2. **Call by reference/address**

   The call by reference method of passing arguments to a function copies the reference of actual arguments into the formal parameters. Inside the function, the reference is used to access the actual argument used in the call.

   Both the actual and formal parameters refer to the same locations, so any changes made inside the function are actually reflected in actual parameters of the caller.

   Example)

```
void swap(int &x, int &y)
{
  int temp;
  temp = x;
  x = y;
  y = temp;
  return;
}
int main ()
{
  int a = 100;
  int b = 200;
  cout << "Before swap, value of a :" << a << endl;
  cout << "Before swap, value of b :" << b << endl;
  swap(a, b);
  cout << "After swap, value of a :" << a << endl;
  cout << "After swap, value of b :" << b << endl;
  return 0;
}
```

OUTPUT:

```
Before swap, value of a :100
Before swap, value of b :200
After swap, value of a :200
After swap, value of b :100
```

3. **Call by pointer**

   The call by pointer method of passing arguments to a function copies the address of an argument into the formal parameter. Inside the function, the address is used to access the actual argument used in the call. This means that changes made to the parameter affect the passed argument.

   To pass the value by pointer, argument pointers are passed to the functions just like any other value.

   Example)

```
void swap(int *x, int *y)
{
  int temp;
  temp = *x;
  *x = *y;
  *y = temp;
  return;
}
int main ()
{
  int a = 100;
  int b = 200;
  cout << "Before swap, value of a :" << a << endl;
  cout << "Before swap, value of b :" << b << endl;
  swap(&a, &b);
  cout << "After swap, value of a :" << a << endl;
  cout << "After swap, value of b :" << b << endl;
  return 0;
}
```

OUTPUT:

Before swap, value of a :100
Before swap, value of b :200
After swap, value of a :200
After swap, value of b :100

Difference b/w Call by Value vs Call by Reference

CALL BY VALUE	CALL BY REFERENCE
While calling a function, we pass values of variables to it. Such functions are known as “Call By Values”.	While calling a function, instead of passing the values of variables, we pass address of variables(location of variables) to the function known as “Call By References.
In this method, the value of each variable in calling function is copied into corresponding dummy variables of the called function.	In this method, the address of actual variables in the calling function are copied into the dummy variables of the called function.
With this method, the changes made to the dummy variables in the called function have no effect on the values of actual variables in the calling function.	With this method, using addresses we would have an access to the actual variables and hence we would be able to manipulate them.
Thus actual values of a and b remain unchanged even after exchanging the values of x and y.	Thus actual values of a and b get changed after exchanging values of x and y.
In call by values we cannot alter the values of actual variables through function calls.	In call by reference we can alter the values of variables through function calls.
Values of variables are passes by Simple technique.	Pointer variables are necessary to define to store the address values of variables.

Note: Since there were no reference variables in C, so there is not any call by reference, but call by pointer in C++ is referred as call by reference in C.

Note: In C++, **both call by pointer and call by reference reflect changes in actual arguments**. Since syntactically, call by reference is simpler, it is preferred. Call by pointer is used over call by reference when reassignment need to be done as references cannot be updated. Call by pointer can also be

used when we require to initialize variable with NULL as it is not possible with reference variables.

Function Return Type

Must Watch:
https://www.youtube.com/watch?v=E8Yh4dw6Diw&list=PL2_aWCzGMAwLZp6LMUKI3cc7pgGsasm2_&index=14

Return Pointer from Functions

We can return pointer from functions but one thing need to be noted is we cannot return pointer to variables in local scope (local to function body), thus we need to declare variables as static so that they are not destroyed(goes out of scope) after function returns from call stack and return pointer to static local variables. { Static Variables have a property of preserving their value even after they are out of their scope. }

```
int* fun()
{
   int A = 10;
   return (&A);
}
int main()
{
   int* p;
   p = fun();
  cout << p << *p << endl;
  return 0;
}
```

This code will give ***Segmentation Fault*** (warning at compile time and error at run-time), since A is local to function *fun().*

Changing above function body like this will run without errors.

```
int* fun()
{
   // Declare a static integer
   static int A = 10;
   return (&A);
}
```

OUTPUT:

```
0x601038
10
```

Return Reference from Functions

A C++ program can be made easier to read and maintain by using references rather than pointers. A C++ function can return a reference in a similar way as it returns a pointer.

When a function returns a reference, it returns an implicit pointer to its return value. This way, a function can be used on the left side of an assignment statement , i.e. **function returning reference**

can act as Lvalue in assignment statement.

Example)

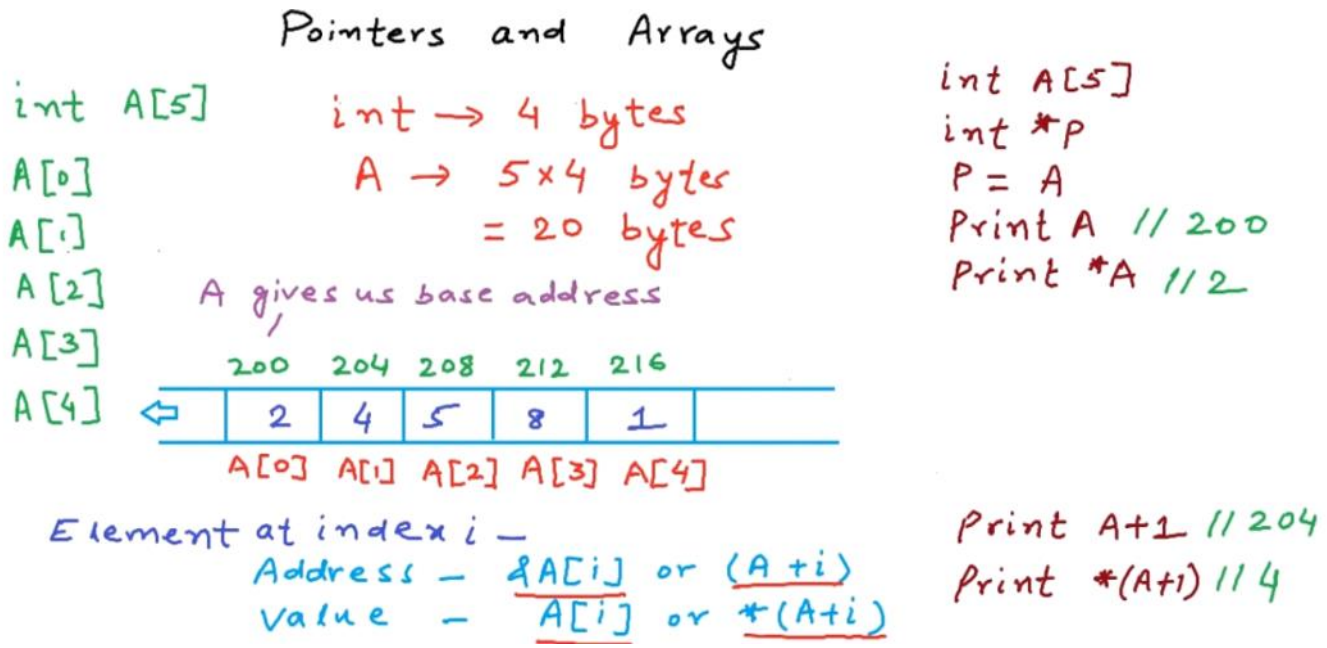
```
int & max(int &x, int &y)
{
    if(x < y) return x;
    else return y;
}
int main()
{
    int a = 6,b = 8;
    max(a,b) = -1;
    return 0;
}
```

Since return type is *int &,* function returns reference to x or y (and not their values).
Function call in Lvalue will yield a reference to either a or b depending on their values.
This assignment statement will assign -1 to b as it is maximum.

Since reference need to be returned, constants like intergal constants (0,1,etc), NULL, etc. cannot be returned.

Also to access reference as Lvalue in calling function, variable to which reference is referring should not be local variable of called function body, otherwise *segmentation fault* will occur.

1D Array & Pointers



Must Watch MyCodeSchool : https://www.youtube.com/watch?v=ASVB8KAFypk&list=PL2_aWCzGMAwLZp6LMUKI3cc7pgGsasm2_&index=6

Array Name as Pointers

An array name acts like a pointer constant. The value of this pointer constant is the address of the first element which is known as the *base address of array*.
For example, if we have an array named arr then **arr** and **&arr[0]** can be used interchangeably.

```
#include<stdio.h>
int SumOfElements(int* A, int size)// "int* A" or "int A[]"  ..it's the same..
{
    int i, sum = 0;

    for(i = 0;i< size;i++)
    {
        sum+= A[i]; // A[i] is *(A+i)
    }
    return sum;
}
int main()
{
    int A[]= {1,2,3,4,5};
    int size = sizeof(A)/sizeof(A[0]);
    int total = SumOfElements(A,size); // A can be used for &A[0]
    printf("Sum of elements = %d\n",total);
    printf("Main - Size of A = %d, size of A[0] = %d\n",sizeof(A),sizeof(A[0]));
}
```

Must Watch MyCodeSchool : https://www.youtube.com/watch?v=CpjVucvAc3g&list=PL2_aWCzGMAwLZp6LMUKI3cc7pgGsasm2_&index=7

Array Decay

The loss of type and dimensions of an array is known as decay of an array.This generally occurs when we pass the array into **function by value or pointer**. What it does is, it sends first address to the array which is a pointer, hence the size of array is not the original one, but the one occupied by the pointer in the memory.

```
// C++ code to demonstrate array decay
#include<iostream>
using namespace std;
```

```
// Driver function to show Array decay
// Passing array by value
void aDecay(int *p)
{
    // Printing size of pointer
    cout << "Modified size of array is by passing by value: ";
    cout << sizeof(p) << endl;
}

// Function to show that array decay happens
// even if we use pointer
void pDecay(int (*p)[7])
{
    // Printing size of array
    cout << "Modified size of array by passing by pointer: ";
    cout << sizeof(p) << endl;
}

int main()
{
    int a[7] = {1, 2, 3, 4, 5, 6, 7,};

    // Printing original size of array
    cout << "Actual size of array is: ";
    cout << sizeof(a) <<endl;

    // Calling function by value
    aDecay(a);

    // Calling function by pointer
    pDecay(&a);

    return 0;
}
```

Output:

```
Actual size of array is: 28
Modified size of array by passing by value: 8
Modified size of array by passing by pointer: 8
```

In the above code, the actual array has 7 int elements and hence has 28 size. But by calling by value and pointer, array decays into pointer and prints the size of 1 pointer i.e. 8 (4 in 32 bit).

How to prevent Array Decay?

A typical solution to handle decay is to pass size of array also as a parameter and not use sizeof on array parameters.

Another way to prevent array decay is to send the array into functions by reference. This prevents conversion of array into a pointer, hence prevents the decay.

Char Array & Pointers

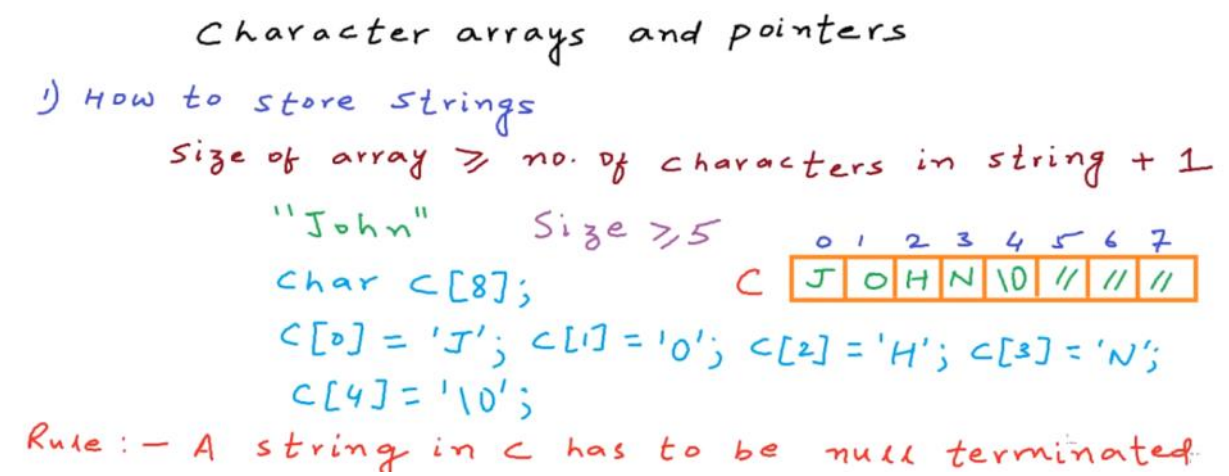
Char Array Part I Must Watch :

https://www.youtube.com/watch?v=Bf8a6IC1dE8&list=PL2_aWCzGMAwLZp6LMUKI3cc7pgGsasm2_&index=8

Char Array Part 2 Must Watch :

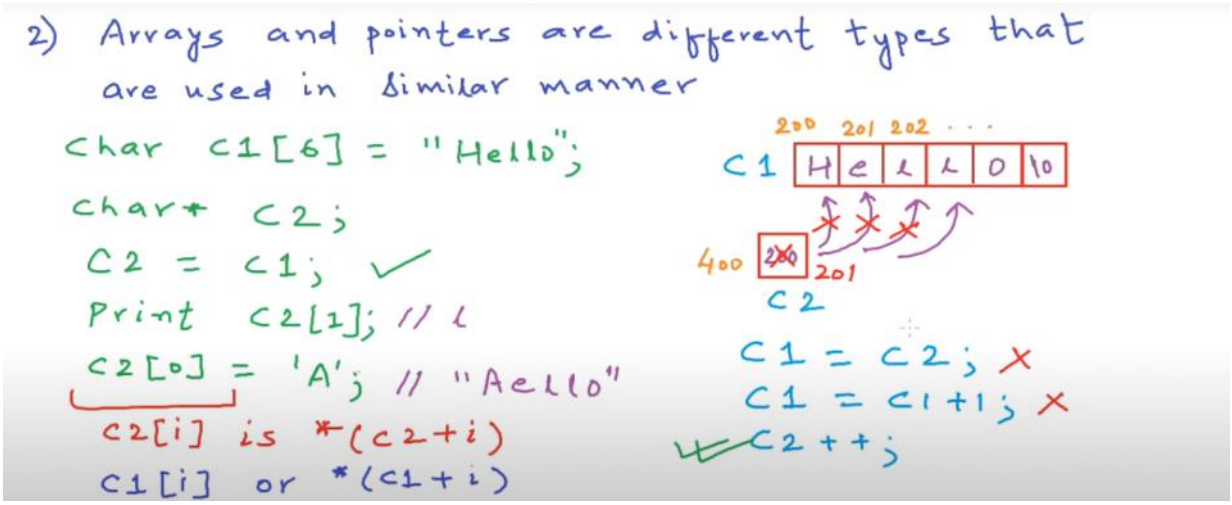
https://www.youtube.com/watch?v=vFZTxvUoZSU&list=PL2_aWCzGMAwLZp6LMUKI3cc7pgGsasm2_&index=9

Char Array & Pointer

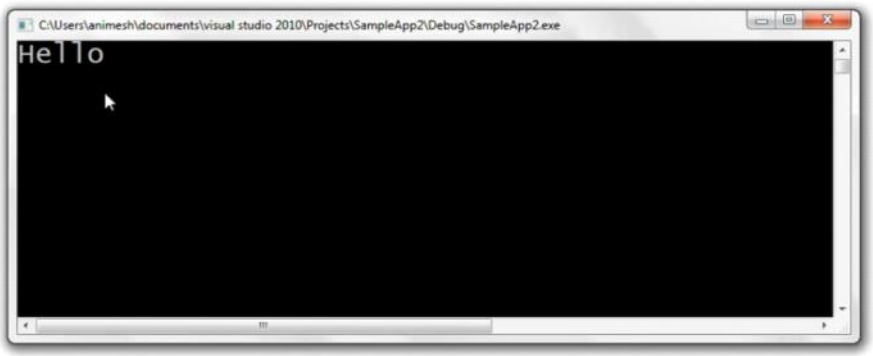


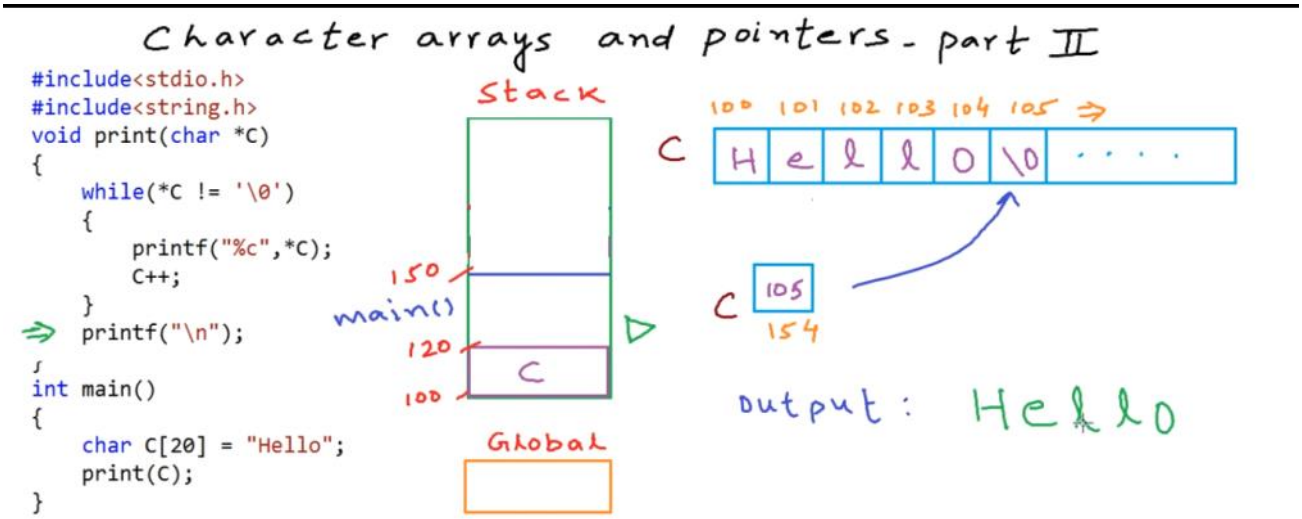

```c
//character arrays and pointers
#include<stdio.h>
#include<string.h>
int main()
{
    char C[] = "JOHN";
    printf("Size in bytes = %d\n",sizeof(C));
    int len = strlen(C);
    printf("Length = %d\n",len);
}
```

```
Size in bytes = 5
Length = 4
```



```
//character arrays and pointers
#include<stdio.h>
void print(char* C)
{
    while(*C != '\0')
    {
        printf("%c",*C);
        C++;
    }
    printf("\n");
}
int main()
{
    char C[20] = "Hello";
    print(C);
}
```





String vs Character Array

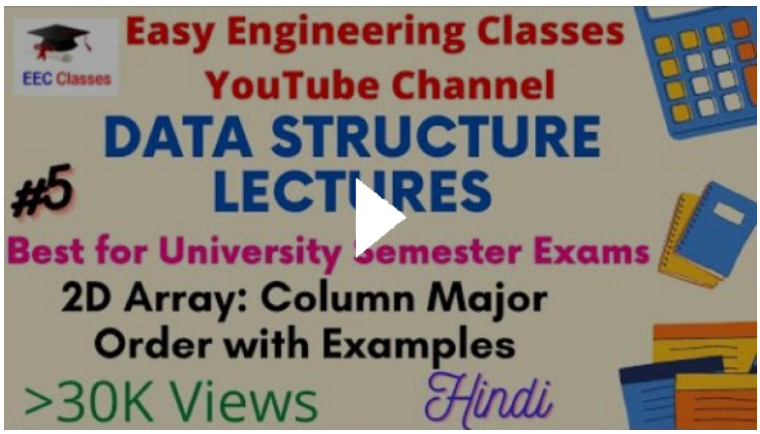
- A character array is simply an **array of characters** terminated by a null character. A string is a **class which defines objects** that be represented as stream of characters.
- Size of the character array has to **allocated statically**, more memory cannot be allocated at run time if required. Unused allocated **memory is wasted** in case of character array. In case of strings, memory is **allocated dynamically**. More memory can be allocated at run time on demand. As no memory is pre-allocated, **no memory is wasted**.
- There is a **threat of array decay** in case of character array. As strings are represented as objects, **no array decay** occurs.
- Implementation of **character array is faster** than std:: string. **Strings are slower** when compared to implementation than character array.
- Character array **do not offer** much **inbuilt functions** to manipulate strings. String class defines **a number of functionalities** which allow manifold operations on strings.

  Refer : https://www.geeksforgeeks.org/char-vs-stdstring-vs-char-c/

2D Array & Pointers

Row Major Address Calculation : [2D Array Representation in Memory – ROW MAJOR ORDER with Example in Hindi and English]

Column Major Address Calculation : [2D Array Representation in Memory – COLUMN MAJOR ORDER with Example in Hindi and English]



Must Watch Part 1: https://www.youtube.com/watch?v=sHcnvZA2u88&list=PL2_aWCzGMAwLZp6LMUKI3cc7pgGsasm2_&index=10

Must Watch Part 2: https://www.youtube.com/watch?v=_j5lhHWkbnQ&list=PL2_aWCzGMAwLZp6LMUKI3cc7pgGsasm2_&index=11

Pointers and multi-dimensional arrays

```
int B[2][3]
```

B[0], B[1] → 1-D arrays of 3 integers

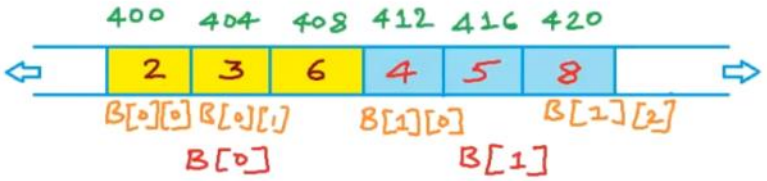

```
int *p = B;   X
```

↓ will return a pointer to 1-D array of 3 integers

```
int (*p)[3] = B;   ✓
```

```
int B[2][3]
```

B[0], B[1] → 1-D arrays of 3 integers

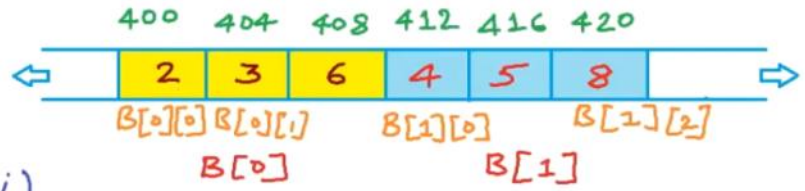



```
int (*p)[3] = B;
Print B or &B[0]          // 400
Print *B or B[0] or &B[0][0]   // 400
Print B+1 or &B[1]        // 412
Print *(B+1) or B[1] or &B[1][0]   // 412
Print *(B+1)+2 or B[1]+2 or &B[1][2]   // 420
Print *(*B+1)             // 3
```

B[0][1]

mycodeschool.com

```
int B[2][3]
```



For 2-D array

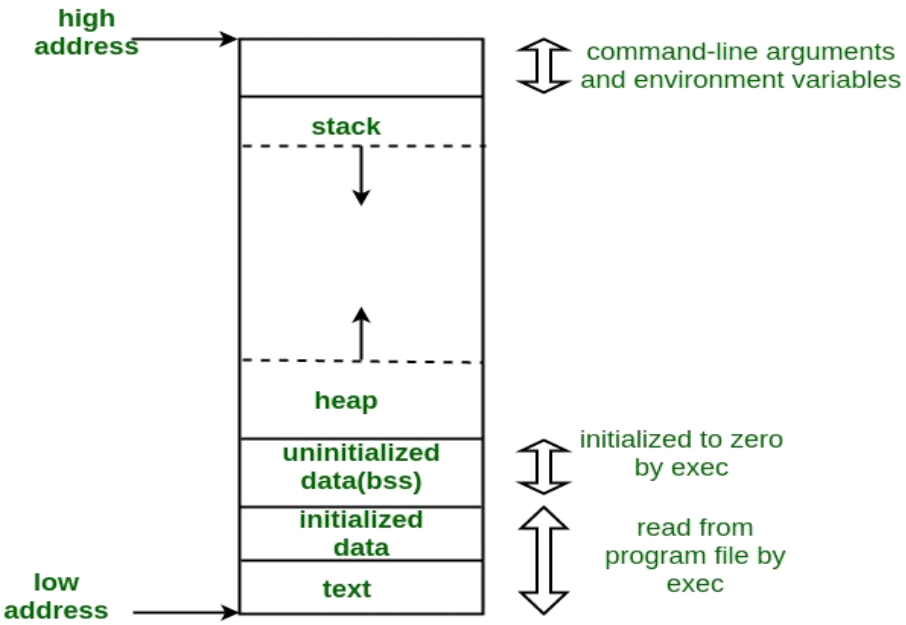
```
B[i][j] = *(B[i]+j)
        = *(*(B+i)+j)
```

Refer: https://www.geeksforgeeks.org/pointer-array-array-pointer/

Memory Layout

A typical memory representation of C program consists of following sections.

1. Text segment
2. Initialized data segment
3. Uninitialized data segment
4. Stack
5. Heap



A typical memory layout of a running process

1. Text Segment:

A text segment , also known as a code segment or simply as text, is one of the sections of a program in an object file or in memory, which contains executable instructions.

As a memory region, a text segment may be placed below the heap or stack in order to prevent heaps and stack overflows from overwriting it.

Usually, the text segment is sharable so that only a single copy needs to be in memory for frequently executed programs, such as text editors, the C compiler, the shells, and so on. Also, the text segment is often read-only, to prevent a program from accidentally modifying its instructions.

2. Initialized Data Segment:

Initialized data segment, usually called simply the Data Segment. A data segment is a portion of virtual address space of a program, which contains the global variables and static variables that are initialized by the programmer.

Note that, data segment is not read-only, since the values of the variables can be altered at run time.

This segment can be further classified into initialized read-only area and initialized read-write area.

For instance the global string defined by char s[] = “hello world” in C and a C statement like int debug=1 outside the main (i.e. global) would be stored in initialized read-write area. And a global C statement like const

char* string = “hello world” makes the string literal “hello world” to be stored in initialized read-only area and the character pointer variable string in initialized read-write area.

Ex: static int i = 10 will be stored in data segment and global int i = 10 will also be stored in data segment

3. Uninitialized Data Segment:

Uninitialized data segment, often called the “bss” segment, named after an ancient assembler operator that stood for “block started by symbol.” Data in this segment is initialized by the kernel to arithmetic 0 before the program starts executing

uninitialized data starts at the end of the data segment and contains all global variables and static variables that are initialized to zero or do not have explicit initialization in source code.

For instance a variable declared static int i; would be contained in the BSS segment.

For instance a global variable declared int j; would be contained in the BSS segment.

4. Stack:

The stack area traditionally adjoined the heap area and grew the opposite direction; when the stack pointer met the heap pointer, free memory was exhausted. (With modern large address spaces and virtual memory techniques they may be placed almost anywhere, but they still typically grow opposite directions.)

The stack area contains the program stack, a LIFO structure, typically located in the higher parts of memory. On the standard PC x86 computer architecture it grows toward address zero; on some other architectures it grows the opposite direction. A “stack pointer” register tracks the top of the stack; it is adjusted each time a value is “pushed” onto the stack. The set of values pushed for one function call is termed a “stack frame”; A stack frame consists at minimum of a return address.

Stack, where automatic variables are stored, along with information that is saved each time a function is called. Each time a function is called, the address of where to return to and certain information about the caller’s environment, such as some of the machine registers, are saved on the stack. The newly called function then allocates room on the stack for its automatic and temporary variables. This is how recursive functions in C can work. Each time a recursive function calls itself, a new stack frame is used, so one set of variables doesn’t interfere with the variables from another instance of the function.

5. Heap:

Heap is the segment where dynamic memory allocation usually takes place.

The heap area begins at the end of the BSS segment and grows to larger addresses from there.The Heap area is managed by malloc, realloc, and free, which may use the brk and sbrk system calls to adjust its size (note that the use of brk/sbrk and a single “heap area” is not required to fulfill the contract of malloc/realloc/free; they may also be implemented using mmap to reserve potentially non-contiguous regions of virtual memory into the process’ virtual address space). The Heap area is shared by all shared libraries and dynamically loaded modules in a process.

Stack vs Heap:

https://www.youtube.com/watch?v=_8-ht2AKyH4&list=PL2_aWCzGMAwLZp6LMUKI3cc7pgGsasm2_&index=12

Memory in a C/C++ program can either be allocated on stack or heap.

Stack Allocation : The allocation happens on contiguous blocks of memory. We call it stack memory allocation because the allocation happens in function call stack. The size of memory to be allocated is known to compiler and whenever a function is called, its variables get memory allocated on the stack. And whenever the function call is over, the memory for the variables is deallocated. This all happens using some predefined routines in compiler. Programmer does not have to worry about

memory allocation and deallocation of stack variables.

Heap Allocation : The memory is allocated during execution of instructions written by programmers. Note that the name heap has nothing to do with heap data structure. It is called heap because it is a pile of memory space available to programmers to allocated and de-allocate. If a programmer does not handle this memory well, memory leak can happen in the program.

Key Differences Between Stack and Heap Allocations

- In a stack, the allocation and deallocation is automatically done by whereas, in heap, it needs to be done by the programmer manually.
- Handling of Heap frame is costlier than handling of stack frame.
- Memory shortage problem is more likely to happen in stack whereas the main issue in heap memory is fragmentation.
- Stack frame access is easier than the heap frame as the stack have small region of memory and is cache friendly, but in case of heap frames which are dispersed throughout the memory so it cause more cache misses.
- Stack is not flexible, the memory size allotted cannot be changed whereas a heap is flexible, and the allotted memory can be altered.
- Accessing time of heap takes is more than a stack.

Comparison Chart:

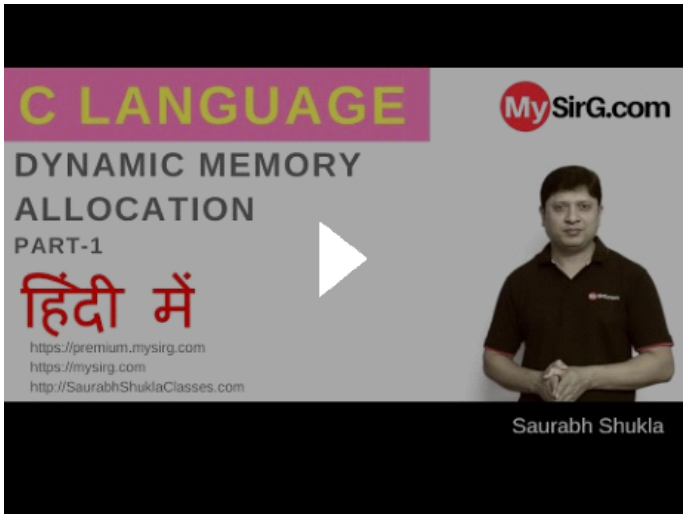
PARAMETER	STACK	HEAP
Basic	Memory is allocated in a contiguous block.	Memory is allocated in any random order.
Allocation and Deallocation	Automatic by compiler instructions.	Manual by programmer.
Cost	Less	More
Implementation	Hard	Easy
Access time	Faster	Slower
Main Issue	Shortage of memory	Memory fragmentation
Locality of reference	Excellent	Adequate
Flexibility	Fixed size	Resizing is possible
Data type structure	Linear	Hierarchical

Reasons and Advantage of allocating memory dynamically:

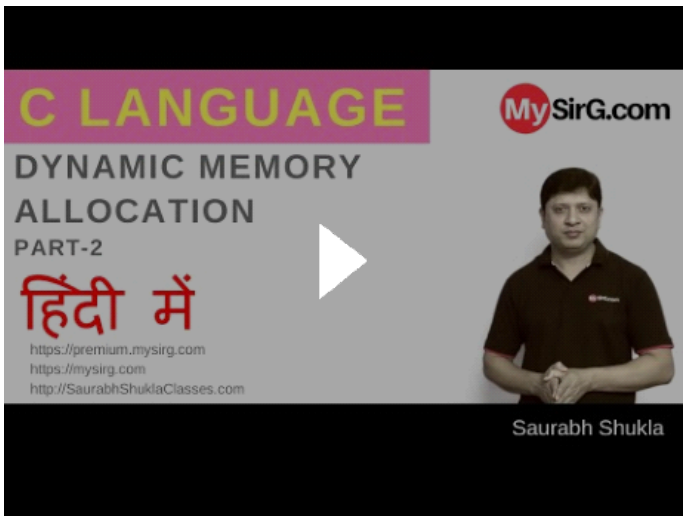
1. When we do not know how much amount of memory would be needed for the program beforehand.
2. When we want data structures without any upper limit of memory space.
3. When you want to use your memory space more efficiently.*Example:* If you have allocated memory space for a 1D array as array[20] and you end up using only 10 memory spaces then the remaining 10 memory spaces would be wasted and this wasted memory cannot even be utilized by other program variables.

4. Dynamically created lists insertions and deletions can be done very easily just by the manipulation of addresses whereas in case of statically allocated memory insertions and deletions lead to more movements and wastage of memory.
5. When you want you to use the concept of structures and linked list in programming, dynamic memory allocation is a must.

Part 1 Saurabh Shukla : Lecture 18 Dynamic Memory Allocation in C Language Part 1 Hindi



Part 2 Saurabh Shukla : Lecture 18 Dynamic Memory Allocation in C language Part 2 Hindi



Malloc Calloc Realloc Free: https://www.youtube.com/watch?v=xDVC3wKjS64&list=PL2_aWCzGMAwLZp6LMUKI3cc7pgGsasm2_&index=13

Dynamic Memory Allocation can be defined as a procedure in which the size of a data structure (like Array) is changed during the runtime.Thus, the mechanism by which storage/memory/cells can be allocated to variables during the run time is called *Dynamic Memory Allocation*.

C provides some functions to achieve these tasks. There are 4 library functions provided by C defined under <stdlib.h> header file to facilitate dynamic memory allocation in C programming. They are:

1. malloc()
2. calloc()
3. free()

4. realloc()

malloc()

“malloc” or **“memory allocation”** method in C is used to dynamically allocate a single large block of memory with the specified size. It returns a pointer of type void which can be cast into a pointer of any form. It initializes each block with default garbage value.

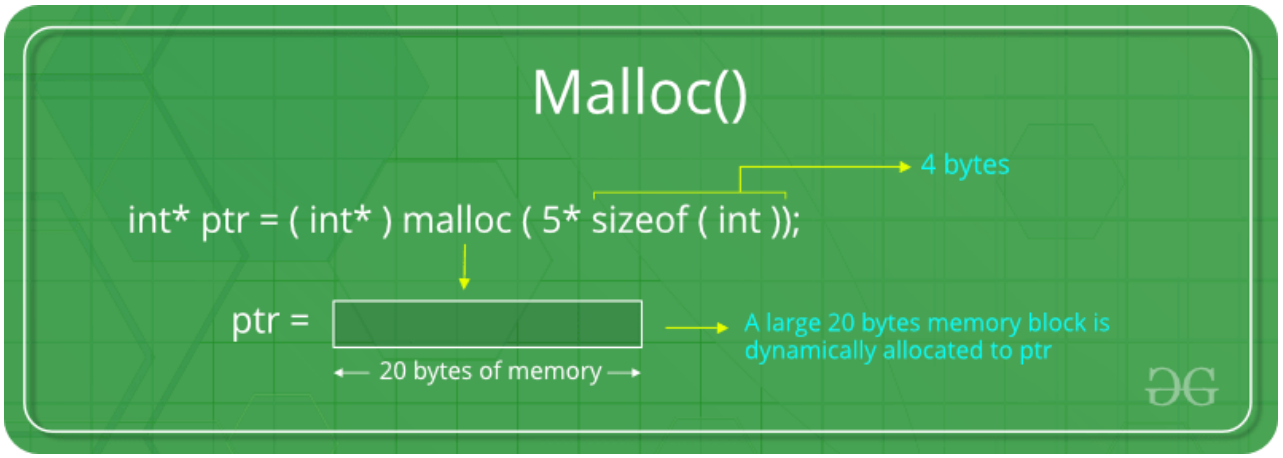
Syntax:

```
ptr = (cast-type*) malloc(byte-size)
```

For Example:

ptr = (int) malloc(100 * sizeof(int));*

Since the size of *int* is 4 bytes, this statement will allocate 400 bytes of memory. And, the pointer *ptr* holds the address of the first byte in the allocated memory. If space is insufficient, allocation fails and returns a NULL pointer.



calloc()

“calloc” or **“contiguous allocation”** method in C is used to dynamically allocate the specified number of blocks of memory of the specified type. It initializes each block with a default value ‘0’.

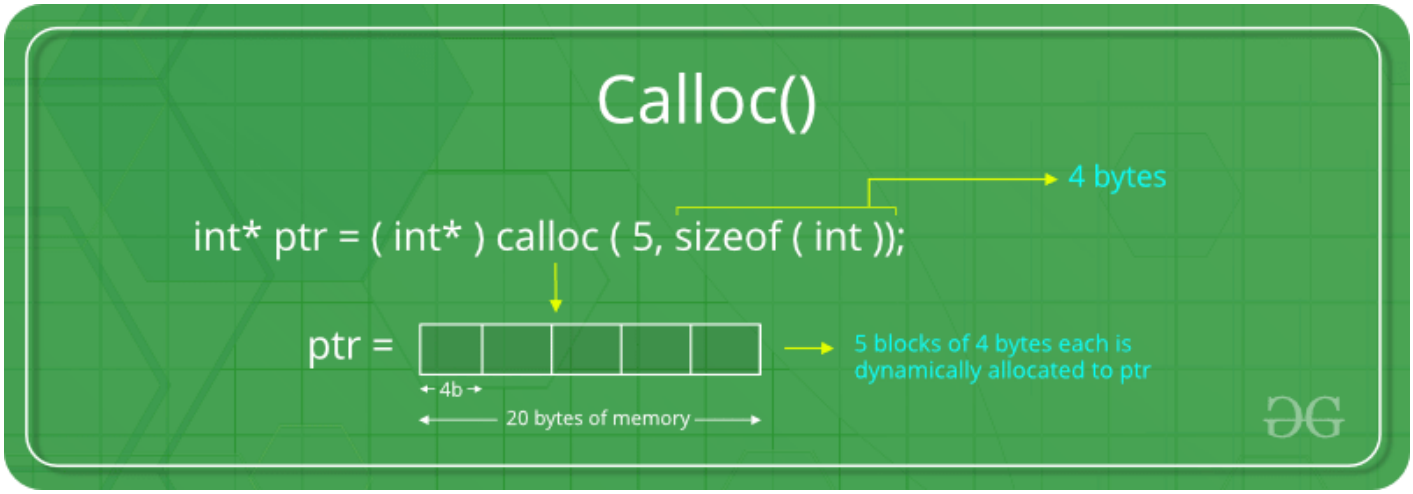
Syntax:

```
ptr = (cast-type*)calloc(n, element-size);
```

For Example:

ptr = (float) calloc(25, sizeof(float));*

This statement allocates contiguous space in memory for 25 elements each with the size of the float. If space is insufficient, allocation fails and returns a NULL pointer.

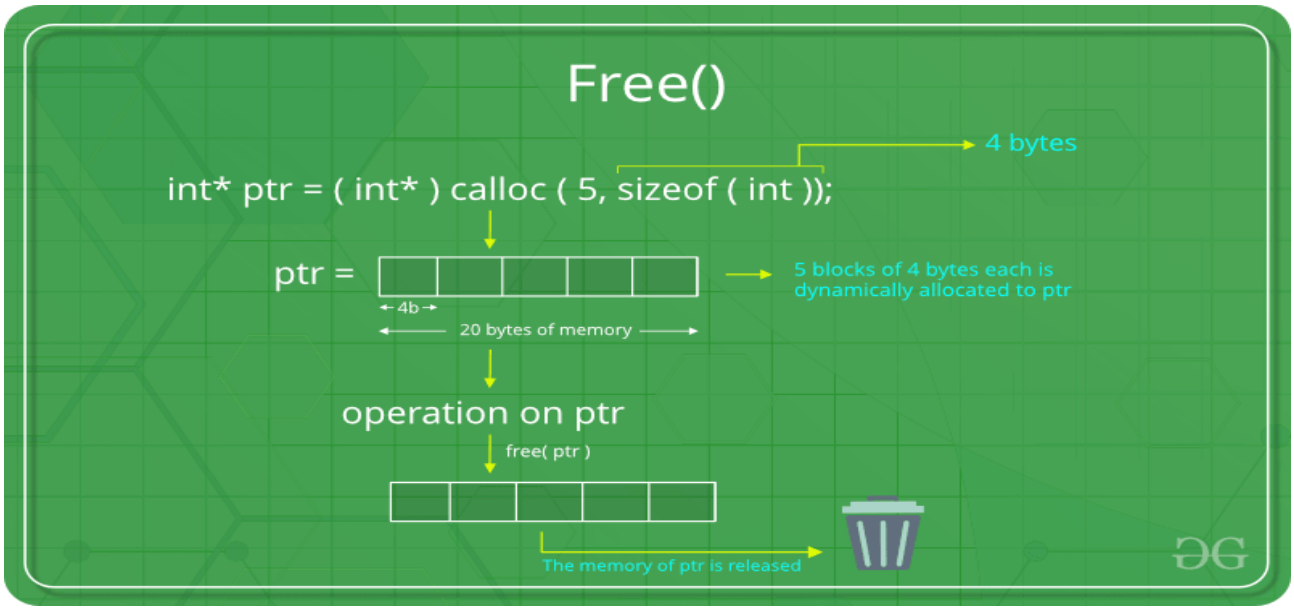


free()

"free" method in C is used to dynamically **de-allocate** the memory. The memory allocated using functions malloc() and calloc() is not de-allocated on their own. Hence the free() method is used, whenever the dynamic memory allocation takes place. It helps to reduce wastage of memory by freeing it.

Syntax:

```
free(ptr);
```



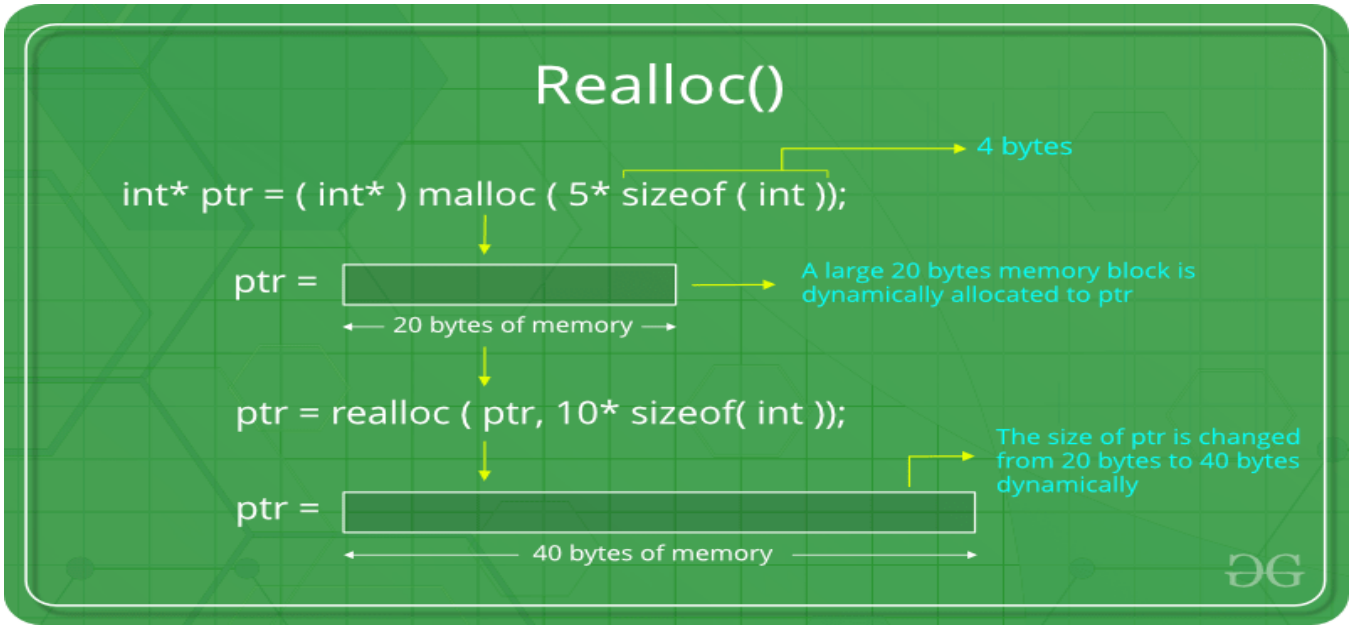
realloc()

"realloc" or **"re-allocation"** method in C is used to dynamically change the memory allocation of a previously allocated memory. In other words, if the memory previously allocated with the help of malloc or calloc is insufficient, realloc can be used to **dynamically re-allocate memory**. re-allocation of memory maintains the already present value and new blocks will be initialized with default garbage value.

Syntax:

```
ptr = realloc(ptr, newSize);
where ptr is reallocated with new size 'newSize'.
```

If space is insufficient, allocation fails and returns a NULL pointer.


Realloc()
4 bytes
int* ptr = (int*) malloc (5* sizeof (int));
ptr =
A large 20 bytes memory block is dynamically allocated to ptr
20 bytes of memory
ptr = realloc (ptr, 10* sizeof(int));
The size of ptr is changed from 20 bytes to 40 bytes dynamically
ptr =
40 bytes of memory

DMA in C++

Dynamic memory allocation in C/C++ refers to performing memory allocation manually by programmer. Dynamically allocated memory is allocated on Heap and non-static and local variables get memory allocated on Stack.

Lecture 16 new and delete in C++ Part 1Hindi

Applications

One use of dynamically allocated memory is to allocate memory of variable size which is not possible with compiler allocated memory except variable length arrays.
The most important use is flexibility provided to programmers. We are free to allocate and deallocate memory whenever we need and whenever we don't need anymore. There are many cases where this flexibility helps. Examples of such cases are Linked List, Tree, etc.

Q) How is it different from memory allocated to normal variables?
R) For normal variables like "int a", "char str[10]", etc, memory is automatically allocated and deallocated. For dynamically allocated memory like "int *p = new int[10]", it is programmers responsibility to deallocate memory when no longer needed. If programmer doesn't deallocate memory, it causes memory leak (memory is not deallocated until program terminates).

Q) How is memory allocated/deallocated in C++?
R) C uses malloc() and calloc() function to allocate memory dynamically at run time and uses free() function to free dynamically allocated memory. C++ supports these functions and also has two operators ***new*** and ***delete*** that perform the task of allocating and freeing the memory in a better and easier way.

new operator

The new operator denotes a request for memory allocation on the Free Store. If sufficient memory is available, new operator initializes the memory and returns the address of the newly allocated and initialized memory to the pointer variable.

- Syntax: *pointer-variable = new data-type;*
  Here, pointer-variable is the pointer of type data-type. Data-type could be any built-in data type including array or any user defined data types including structure and class.

  Example:
  // Pointer initialized with NULL and then request memory for the variable
  *int *p = NULL;*
  p = new int;

  OR
  // Combine declaration of pointer and their assignment
  *int *p = new int;*

- Initialize memory: We can also initialize the memory using new operator:
  pointer-variable = new data-type(value);

  Example:
  *int *p = new int(25);*
  *float *q = new float(75.25);*

- Allocate block of memory: new operator is also used to allocate a block(an array) of memory of type data-type.
  pointer-variable = new data-type[size];
  where size(a variable) specifies the number of elements in an array.

  Example: *int *p = new int[10];*
  Dynamically allocates memory for 10 integers continuously of type int and returns pointer to the first element of the sequence, which is assigned to p(a pointer). p[0] refers to first element, p[1] refers to second element and so on.

Normal Array Declaration vs Using new

There is a difference between declaring a normal array and allocating a block of memory using new. The most important difference is, normal arrays are deallocated by compiler (If array is local, then deallocated when function returns or completes). However, dynamically allocated arrays always remain there until either they are deallocated by programmer or program terminates.

What if enough memory is not available during runtime?

If enough memory is not available in the heap to allocate, the *new* request indicates failure by throwing an exception of type *std::bad_alloc*, unless "nothrow" is used with the new operator, in which case it returns a NULL pointer. Therefore, it may be good idea to check for the pointer variable produced by new before using it program.

```
int *p = new(nothrow) int;
if (!p)
{
   cout << "Memory allocation failed\n";
}
```

malloc vs new

Following are the differences between malloc() and operator new.:

1. **Calling Constructors:** new calls constructors, while malloc() does not. In fact primitive data types (char, int, float.. etc) can also be initialized with new. For example, below program prints 10.

```
#include<iostream>
using namespace std;
int main()
{
    // Initialization with new()
```

```
    int *n = new int(10);
    cout << *n;
    getchar();
    return 0;
}
```

Output:

```
10
```

2. **operator vs function:** new is an operator, while malloc() is a function. Thus new (&delete) can be overloaded like any other operator.
3. **return type:** new returns exact data type, while malloc() returns void *.
4. **Failure Condition:** On failure, malloc() returns NULL where as new throws *bad_alloc* exception.
5. **Memory:** In case of new, memory is allocated from free store where as in malloc() memory allocation is done from heap.
6. **Size:** Required size of memory is calculated by compiler(automatically internally thus we need not use *sizeof* operator) for new, whereas we have to manually calculate size for malloc().
7. **Buffer Size:** malloc() allows to change the size of buffer using realloc() while new doesn't.
8. **Dynamic Initialization:** It is possible to initialize the object while creating the memory space using *new* but not possible with *malloc().*

NEW	MALLOC
CALLS CONSTRUCTOR	DOES NOT CALLS CONSTRUCTORS
IT IS AN OPERATOR	IT IS A FUNCTION
RETURNS EXACT DATA TYPE	RETURNS VOID *
ON FAILURE, THROWS	ON FAILURE, RETURNS NULL
SIZE IS CALCULATED BY COMPILER	SIZE IS CALCULATED MANUALLY

Delete Operator

Since it is programmer's responsibility to deallocate dynamically allocated memory, programmers are provided delete operator by C++ language.

Syntax: *delete pointer-variable;*
Here, pointer-variable is the pointer that points to the data object created by new.

Examples:
delete p;
delete q;

To free the dynamically allocated array pointed by pointer-variable, use following form of delete:
// Release block of memory pointed by pointer-variable
delete[] pointer-variable;

Example:
// It will free the entire array pointed by p.
delete[] p;

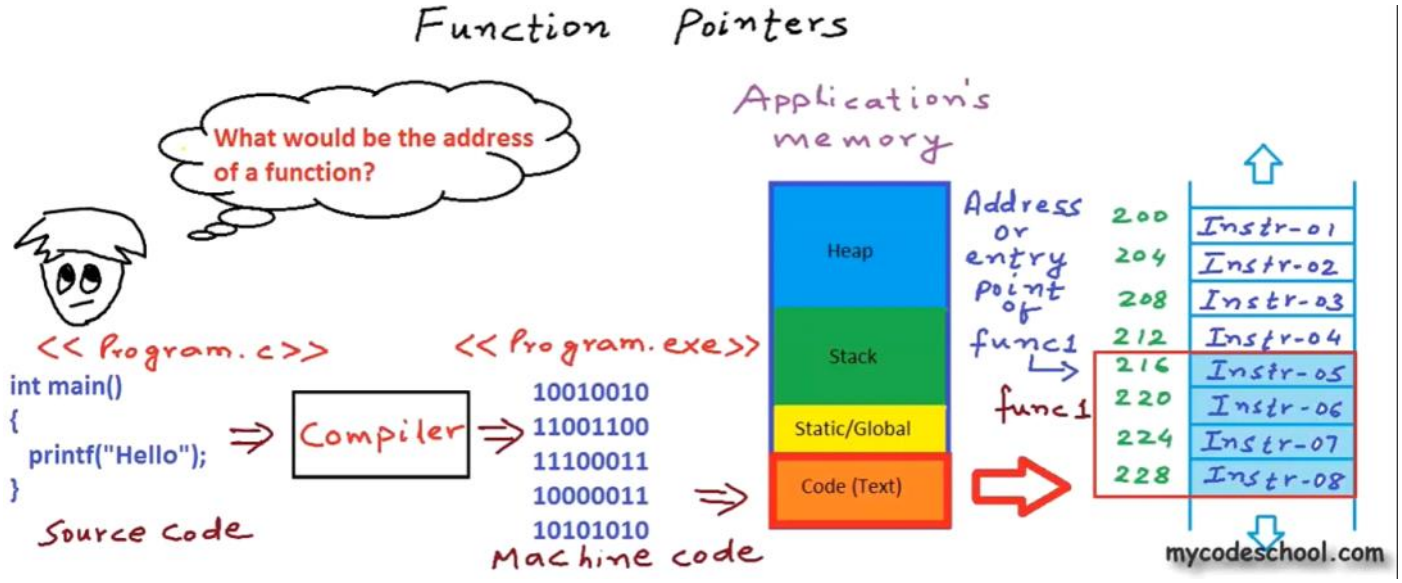
Note: In C++, delete operator should only be used either for the pointers pointing to the memory

allocated using new operator or for a NULL pointer, and free() should only be used either for the pointers pointing to the memory allocated using malloc() or for a NULL pointer.

Function Pointer

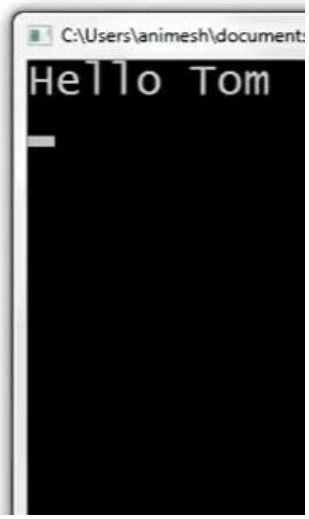
Must Watch Part 1:
https://www.youtube.com/watch?v=ynYtgGUNelE&list=PL2_aWCzGMAwLZp6LMUKI3cc7pgGsasm2_&index=15

Must Watch Part 2:
https://www.youtube.com/watch?v=sxTFSDAZM8s&list=PL2_aWCzGMAwLZp6LMUKI3cc7pgGsasm2_&index=16




```c
//Function Pointers in C/C++
#include<stdio.h>

int Add(int a,int b)
{
    return a+b;
}
int main()
{
    int c;
    int (*p)(int,int);
    p = Add; // function name will return us pointer
    c = p(2,3); //de-referencing and executing the function.
    printf("%d",c);
}
```

```c
//Function Pointers in C/C++
#include<stdio.h>
void PrintHello(char *name)
{
    printf("Hello %s\n",name);
}
int Add(int a,int b)
{
    return a+b;
}
int main()
{
    void (*ptr)(char*);
    ptr = PrintHello;
    ptr("Tom");
}
```




```c
//Function Pointers and callbacks
#include<stdio.h>
void A()
{
    printf("Hello");
}
void B(void (*ptr)()) // function pointer as argument
{
    ptr(); //call-back function that "ptr" points to
}
int main()
{
    B(A); // A is callback function.
}
```

```c
#include<stdio.h>
int compare(int a,int b)
{
    if(a > b) return 1;
    else return -1;
}
void BubbleSort(int *A,int n,int (*compare)(int,int)) {
    int i,j,temp;
    for(i =0; i<n; i++)
        for(j=0; j<n-1; j++) {
            if(compare(A[j],A[j+1]) > 0) { //compare A[j] with A[j+1] and SWAP if needed
                temp = A[j];
                A[j] = A[j+1];
                A[j+1] = temp;
            }
        }
}
int main() {
    int i, A[] ={3,2,1,5,6,4};
    BubbleSort(A,6,compare);
    for(i =0;i<6;i++) printf("%d ",A[i]);
}
```

Like pointers which point to address of another variables, Function pointer point to memory address of another functions (functions are given memory address in Code/Text section of RAM).

Q) Why do we need an extra bracket around function pointers like fun_ptr in above example?
R) If we remove bracket, then the expression "*void (*fun_ptr)(int)*" becomes "*void *fun_ptr(int)*" which is declaration of a ***function that returns void pointer*** and not function pointer.

- Unlike normal pointers, a function pointer points to code, not data. Typically a function pointer stores the start of executable code.

- Unlike normal pointers, we do not allocate de-allocate memory using function pointers.
  A function's name can also be used to get functions' address.
  For example, in the below program, we have removed address operator '&' in assignment. We have also changed function call by removing *, the program still works.

```c
// A normal function with an int parameter and void return type
void fun(int a)
{ printf("Value of a is %d\n", a); }
int main()
{
   void (*fun_ptr)(int) = fun;  // & removed
   fun_ptr(10);  // * removed
   return 0;
}
```

Output:

Value of a is 10

- Like normal pointers, we can have an array of function pointers.

```
void add(int a, int b)
{  printf("Addition is %d\n", a+b); }
void subtract(int a, int b)
{  printf("Subtraction is %d\n", a-b); }
void multiply(int a, int b)
{  printf("Multiplication is %d\n", a*b); }

int main()
{
    // fun_ptr_arr is an array of function pointers
    void (*fun_ptr_arr[])(int, int) = {add, subtract, multiply};
    unsigned int ch, a = 15, b = 10;

    printf("Enter Choice: 0 for add, 1 for subtract and 2 for multiply\n");
    scanf("%d", &ch);

    if (ch > 2) return 0;

    (*fun_ptr_arr[ch])(a, b);

    return 0;
}
```

- Like normal data pointers, a function pointer can be passed as an argument and can also be returned from a function.
  For example, consider the following C program where wrapper() receives a void fun() as parameter and calls the passed function.

```
// Two simple functions
void fun1() { printf("Fun1\n"); }
void fun2() { printf("Fun2\n"); }

// A function that receives a simple function
// as parameter and calls the function
void wrapper(void (*fun)())
{
    fun();
}

int main()
{
    wrapper(fun1);
    wrapper(fun2);
    return 0;
}
```

- Many object oriented features in C++ are implemented using function pointers in C. For example virtual functions. Class methods are another example implemented using function pointers.

Const Pointer

The qualifier **const** can be applied to the declaration of any variable to specify that its value will not be changed (which depends upon where const variables are stored, we may change the value of const variable by using pointer). const keyword applies to whatever is immediately to its left. If there is nothing to its left, it applies to whatever is immediately to its right.

1. **Pointer to Variable**

   int x = 5;
   int *Ptr = &x;
   Ptr++; => Valid
   (*Ptr)++; => Valid

   Note: If we make x as constant but don't make ptr as pointer to const, we can modify the value of x using ptr.

   const *int x = 5;*
   *int *Ptr = &x;*
   This type of assignment of &x (const int*) to ptr (int*) is known as **Down Qualification** in C++, It causes compiler to implicitly remove const-ness from the expression &x. It is not allowed in C++, it will compile successfully but may give unexpected behavior at run-time.
   Ptr++; => Valid Pointer Arithmetic
   *x++; => **Invalid**, cannot update x using const x only.*
   *(*Ptr)++; => **Valid,** will update x to 6, even though x is const,*
   because pointer is not made to point to const

2. **Pointer to Const Variable**

   Pointer to constant can be declared in following two ways:

   const int x = 5;

   *const int *ptr = &x;*
   OR { const and int can be used interchangeabely in any order }
   *int const * ptr = &x;*

   *(*ptr)++; => **Invalid ,** because pointer is made to point to const*
   *x++; => **Invalid**, because variable is itself made const*
   ptr++; => Valid, pointer arithmetic is valid because pointer is itself not const.
   ptr = &y; => Valid, pointer reassignment/updation is valid because pointer is itself not const.

   We can change the pointer to point to any other integer variable and can also do pointer arithmetic, but cannot change the value of the object (entity) pointed using pointer *ptr*.
   The pointer is stored in the read-write area (stack in the present case). The object pointed may be in the read-only or read-write area i.e. object pointed can itself be const or non-const.

   Eg) *int x = 5;*

const *int *ptr = &x;*

This type of assignment of int* (&x) to const int* (ptr) is known as ***Up Qualification*** in C++.
It is valid in C++ and will run successfully. We cannot modify data of x using pointer to const but can modify data of x using x itself.

*x++; => **Valid** , as x is itself not const*

*(*ptr)++; => **Invalid**, as ptr is made to point to const*

3. Const Pointer to Variable

Syntax:

int x = 5;

*int *const ptr = &x;*

Above declaration is a constant pointer to an integer variable, means we can change the value of object pointed by pointer, but cannot change the pointer to point another variable (pointer reassignment and pointer arithmetic is not possible).

*ptr = &y; => **Invalid** Reassignment*

*ptr++; => **Invalid** Pointer Arithmetic*

*(*ptr)++ or *ptr = 6; => Valid*

Note: **Dynamic Initialization of const pointer is also not possible** i.e. they need to be initialized in the same line in which they are declared.

int *const ptr;

ptr = &x; => *Invalid*

```
namespace Box1
{
   int a = 4;
}
namespace Box2
{
   int a = 13;
}

int main ()
{
   int a = 16;
   Box1::a;
   Box2::a;
   cout << a;
```

Screen clipping taken: 11/28/2020 1:18 PM

4. Const Pointer to Const Variable

Syntax:

```
const int x = 5;
const int *const ptr = &x;
```

Above declaration is a constant pointer to a constant variable which means Neither pointer arithmetic or reassignment is possible , nor updation of value using dereferencing operator is possible. Dynamic initialization need to be done.

*(*ptr)++; OR *ptr = 6; => **Invalid***
*ptr++; => **Invalid***
*ptr = &y; => **Invalid***

Wild Pointers

What is Wild Pointer in C Language Hindi

There is always a fixed memory (2^64 bytes for 64-bit architecture) allocated to a C/C++ program. Initially, all of the memory is in free state. As variables are declared, some memory from free state gets consumed and comes in consumed state.
Uninitialized pointers are known as ***wild pointers*** because they point to some arbitrary memory location (from free state) and may cause a program to crash or behave badly.
A pointer which has not been initialized to anything (not even NULL) is known as wild pointer. The pointer may be initialized to a non-NULL garbage value that may not be a valid address.

```
int main()
{
  int *p;  /* wild pointer */
   /* Some unknown memory location is being corrupted.
   This should never be done. They should be initialized with NULL */
   // p will contain some random (may not be even valid) memory address which contain some garbage data.
  *p = 12;
    // We are dereferencing memory address which is not consumed (i.e. in free state) & it can
    give unexpected behavior

  int a = 10;
  p = &a;  /* p is not a wild pointer now*/
  *p = 12; /* This is fine. Value of a is changed */

}
```

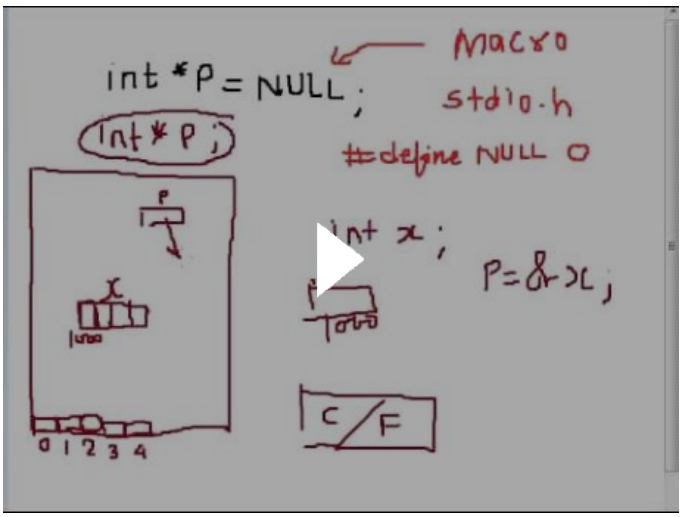
If we want pointer to a value (or set of values) without having a variable for the value, we should explicitly allocate memory and put the value in allocated memory (using *new* operator) in heap, or initialize pointer with NULL (or nullptr).

```
int *p = NULL;
    OR
int *p = new int;
// Although accessing data using *p will even now also have some garbage value but accessing the
memory using p will now point to some valid memory address which is not in free state (in stack),
```

but allocated dynamically in heap.

NULL Pointer

What is NULL Pointer in C Language Hindi



NULL Pointer is a pointer which is pointing to nothing.
Some of the most common use cases for NULL are

a) To initialize a pointer variable when that pointer variable isn't assigned any valid memory address yet (to **avoid wild pointers**)
b) To check for a null pointer before accessing any pointer variable. By doing so, we can **perform error handling** in pointer related code e.g. dereference pointer variable only if it's not NULL.
c) To **pass a null pointer to a function argument** when we don't want to pass any valid memory address.

Syntax:
int *ptr = NULL;
*cout << ptr; => 0 as using NULL ptr, we point to **0th byte** memory address.*
cout << &ptr;
=> Some Memory Address will be given to pointer itself eg "0x61ff0c"
*cout << *ptr;*
*=> **Invalid,** since ptr point to nothing, it should not be deferenced before checking for NULL. Although it may compile , but will give unexpected behavior at run-time.*
cout << sizeof(NULL);
=> Since ***NULL is defined as ((void*)0)***, we can think of NULL as a special pointer and its size would be equal to any pointer. If the pointer size of a platform is 4 bytes, the output of the above program would be 4. But if pointer size on a platform is 8 bytes, the output of the above program would be 8.

Important Points

- NULL vs Uninitialized pointer – An uninitialized pointer stores an undefined value. A null pointer stores a defined value, but one that is defined by the environment to not be a valid address for any member or object.
- NULL vs Void Pointer – Null pointer is a value, while void pointer is a type

nullptr Pointer

Consider the following program:

```
// function with integer argument
int fun(int N)   { cout << "fun(int)"; }

// Overloaded function with char pointer argument
int fun(char* s)  { cout << "fun(char *)"; }

int main()
{
    // Ideally, it should have called fun(char *),
    // but it causes compiler error.=> call of overloaded 'fun(NULL)' is ambiguous
    fun(NULL);
}
```

NULL is typically defined as (void *)0 and conversion of NULL to integral types is allowed. So the function call fun(NULL) becomes ambiguous.
In the above program, if we replace ***NULL*** with ***nullptr***, we get the output as “fun(char *)”.

nullptr is a keyword that can be used at all places where NULL is expected. Like NULL, nullptr is implicitly convertible and comparable to any pointer type. **Unlike NULL, it is not implicitly convertible or comparable to integral types.**

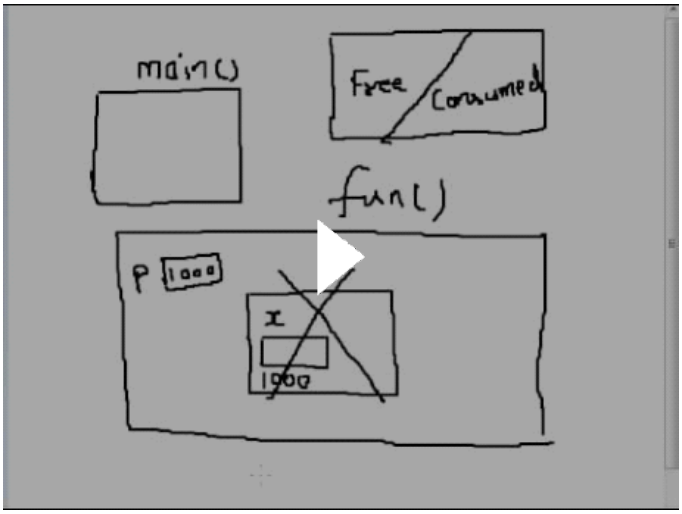
```
// This program compiles (may produce warning)
#include<stdio.h>
int main()
{
   int x = NULL;
}

// This program does NOT compile and give compile time error
#include<stdio.h>
int main()
{
   int x = nullptr;
}
```

Note: **nullptr is convertible to bool.**

```
int main()
{
   int *ptr = nullptr;

   // Below line compiles
   if (ptr) { cout << "true"; }
   else { cout << "false"; }
}
```

Dangling Pointer

What is dangling Pointer in C language Hindi



A pointer pointing to a memory location that has been deleted (or freed) is called dangling pointer. There are three different ways where Pointer acts as dangling pointer:

1. **De-allocation of memory**

```
// Deallocating a memory pointed by ptr causes dangling pointer
#include <stdlib.h>
#include <stdio.h>
int main()
{
    int *ptr = new int;

    // After below free call, ptr becomes a dangling pointer
    free(ptr);  // or delete ptr;

    ptr = NULL;  // No more a dangling pointer
}
```

2. **Function Call**

```
// The pointer pointing to local variable becomes
// dangling when local variable is not static.
int *fun()
{
    // x is local variable and goes out of scope after an execution of fun() is over.
    int x = 5;
    return &x;
}

int main()
{
    int *p = fun();
    fflush(stdin);

    // p points to something which is not valid anymore
    printf("%d", *p);
    return 0;
}
```

Output:

```
A garbage Address
```

Note: The above problem doesn't appear (or p doesn't become dangling) if x is a static variable.

```
// The pointer pointing to local variable doesn't
// become dangling when local variable is static.
#include<stdio.h>

int *fun()
{
    // x now has scope throughout the program
    static int x = 5;
    return &x;
}

int main()
{
    int *p = fun();
    fflush(stdin);
    // Not a dangling pointer as it points to static variable.
    printf("%d",*p);
}
```

Output:

```
 5
```

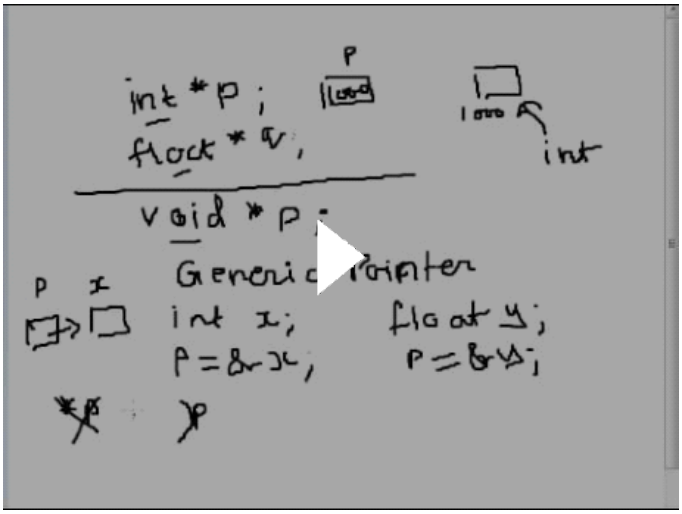
Note: Similarly, we can have a ***dangling reference***, by returning by reference in above example instead of return by pointer.

3. **Variable goes out of scope**

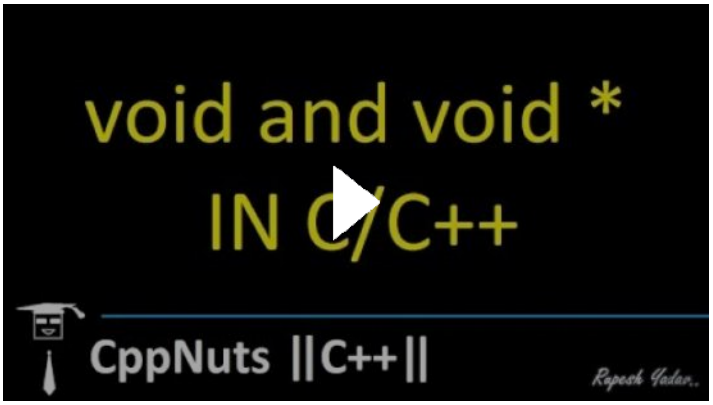
```
void main()
{
   int *ptr;
   .....
   .....
   {
       int ch;
       ptr = &ch;
   }
   .....
   // Here ptr is dangling pointer
}
```

Void Pointer (Generic Pointer)

What is void pointer in C language Hindi



[void and void Pointer In C/C++]



Void pointer (void *) is a specific pointer type , a pointer that points to some data location in storage, which doesn't have any specific type. i.e. A void pointer can hold address of any type and can be typcasted to any type.

If we assign address of char data type to void pointer it will become char Pointer, if int data type then int pointer and so on.

Any pointer type is convertible to a void pointer hence it can point to any value. Although, we cannot directly use value at that memory address using dereferencing(*) with void pointer as we must give explicit type conversion.

Example)

```
int main()
{
    int x = 4;
    float y = 5.5;

    //A void pointer
    void *ptr;
    ptr = &x;

    // (int*)ptr - does type casting of void
    // *((int*)ptr) dereferences the typecasted
    // void pointer variable.
    printf("Integer variable is = %d", *( (int*) ptr) );
```

```
    // void pointer is now float
    ptr = &y;
    printf("\nFloat variable is= %f", *( (float*) ptr) );

    return 0;
}
```

Output:

```
Integer variable is = 4
Float variable is= 5.500000
```

- void pointer is also known as ***Universal Pointer*** or ***Generic Pointer***.
- void pointers ***cannot be dereferenced***. It can however be done using typecasting the void pointer
- **Pointer arithmetic is not possible** on pointers of void due to lack of concrete value and thus size. However, **in GNU C (g++)** it is allowed by considering **the size of void is 1.**
- Any data type can be converted into void* **except function pointer, const or volatile**.

Uses of Void Pointers

1) malloc() and calloc() return void * type and this allows these functions to be used to allocate memory of any data type (just because of void *)

```
int main(void)
{
  // Note that malloc() returns void * which can be typecasted to any type like int *, char *, ..
  int *x = malloc(sizeof(int) * n);
}
```

Note that the above program compiles in C, but doesn't compile in C++. In C++, we must explicitly typecast return value of malloc to (int *) i.e. *int*x = (int*)(malloc(sizeof(int)))*;

2) void pointers in C are used to implement generic functions in C. For example) compare function which is used in qsort().

Pointers to Objects of Class (Object Pointer)

A pointer that contains address of an object of a class is known as object pointer.
Just like a normal pointer which points to primitive data types, object pointer points to an object of a class. Also as with all pointers, you must initialize the pointer before using it.

The . (dot) operator and the -> (arrow) operator are used to reference individual members of classes, structures, and unions. They are known as *member access operators*. The dot operator is applied to the actual object. The arrow operator is used with a pointer to an object.

Example) Consider a class Student with data member *name* and member function *getname()*.

```
Student Obj;
Student *Ptr = &Obj;

// Accessing data members and member functions by an object using dot operator (.)
Obj.name = "Archit";
Obj.getname();

// Accessing data members and member functions by a pointer to object using arrow operator (->)
Ptr->name = "Archit";
Ptr->getname();

// Note: We can also use dot operator(.) with pointer to object but they need to be dereferenced first
inside paranthesis () because dot operator (.) has higher precedence than indirection operator (*).
(*Ptr).name = "Archit";
(*Ptr).getname();
```

Pointers to Members of Class (Member Pointers)

It is possible to take the address of a member of a class and assign it to a pointer. The address of a member can be obtained by applying the operator & to a *"fully qualified"* class member name. A *class member pointer* can be declared using the operator **::*** with the class name.

Example)

```
Class A
{
    public:
    int x;
    int getx();
};
int A::* ptr = &A :: x;  => Pointer to data member x of class A.
```

The pointer ptr created acts like a class member which must be invoked with a class object.

- ***::**** operator is known as '*pointer to member access operator*' of A class.
- ***&A::x*** means '*address of member x of class A*'.

Note: Membership label (A::) must need to be used with x and pointer name both.

Now this pointer *ptr* can be used to access the member *x* inside member functions (or friend functions).

Example) Consider an object *obj* of class A. We can access x using pointer to member ptr in the following way:

*cout << obj.*ptr;*
cout << obj.x; => Both ways are same

Similarly, we can declare a pointer to object *obj* of class A *pobj,* and data member x can be accessed using pointer to object and pointer to data member in following way:

```
A* pobj = &obj; // Pointer to Object or Object Pointer
cout << pobj -> *ptr;
cout << pobj -> x; => Both ways are same
```

Thus **ptr* act like a member name of class A.

The ***deferencing operator ->**** is used to access a member using pointers to both object and the member. The ***deferencing operator .**** is used when object itself is used with pointer to member (or data member pointer).

Note: Just like pointer to data members of a class, we can create pointer to member functions of a class , which will become *member function pointers* for the class.

Syntax: *return_type (class_name::*ptr_name) (argument_type) = &class_name::function_name;*
Example: *int (A::*fptr) () = &A::getx;*

Note:

- You can change the value and behaviour of these pointers on runtime. That means, you can point it to other member function or member variable.
- To have pointer to data member and member functions you need to make them public.

This Pointer

This Pointer in C++ (HINDI/URDU)

This Pointer In C++

C++ uses keyword *this* to represent the pointer to calling object that calls the member functions.

This pointer is **automatically passed to a member function** when it is called using object and dot operator (.) { or using pointer to object and arrow operator (->) }

The pointer *this* acts as an **implicit (or hidden) argument to all the member functions**, thus **local (in scope) to the member function** being called.

It is used to refer to the caller object itself and thus it **contains the address of the caller object**.

Also, it is passed as const argument (**const pointer** not pointer to const) implicitly, so that it cannot be modified (**cannot be made to point to a different object**) by the programmer in the function body but it can modify the value of data members of the caller object using arrow operator (->) or dereferencing operator (*) with dot operator (.) .

Hence, it is a ***local const object pointer***.

Note: Since it is passed implicitly to only member functions, it is **not passed to the friend functions**.
Also, since it is used to contain the address of caller object, and static member functions are not called using objects, hence **this pointer is not passed to static member functions** also.
Similarly, it **cannot be used to get or set the value of static data members** of the class since they are accessed directly using classname (using membership label) and not object.

Note: **If the member function is declared as const** (const member function), then this pointer which is passed implicitly is also pointer to const. Thus **this pointer becomes const pointer pointing to const object** in such case.

Working & Use Cases)

```
class A
{
    public:
    int x = 0;
    void setx(int x)
    {
        this->x = x;      // or it can be written as (*this).x = x;
        // USE 1. To resolve name conflicts when formal arguments have same name as data members
        of the class, this pointer is used to refer to the caller object's data member x.
    }
    A&  getmax(A *obj)
    {
        if(obj.x > this.x)
            return obj;         // Return Argument Object
        else return *this;    // Return Caller Object
        // USE 2. To return reference to the current caller object (object this pointer points to)
    }
};
A Obj1, Obj2, Obj3;
Obj1.setx(5);
Obj2.setx(10);
Obj3 = Obj1.getmax(&Obj2);
```

Internally, function calls are replaced by the following statements:

```
    Obj1 = A::setx(&Obj1, 5);
    Obj3 = A::getmax(&Obj1, &Obj2);
```

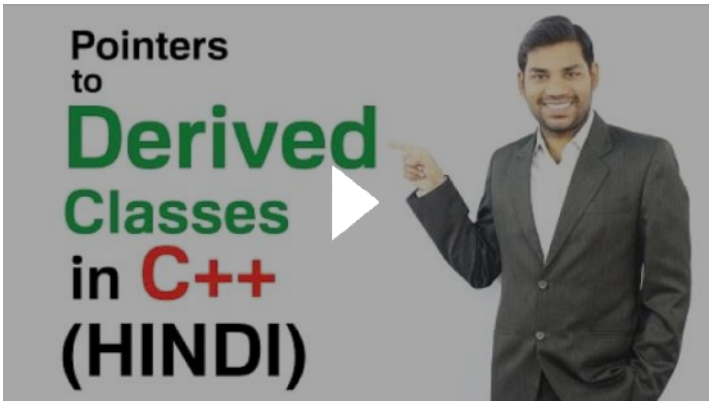
And the function definitions are replaced by the following statements:

```
    void setx(A * const this, int x);
    A& getmax(A * const this, A & obj);
```

Base Class Pointer to Derived Class Object

Must Watch: Abdul Bari Video + Below Video

Pointers To Derived Classes in C++ (HINDI)



We can not only make pointers of type Base class to point to Base class objects, but also to Derived class objects.

Pointers to objects of base class are type-compatible with pointers to objects of derived class. Therefore, a single pointer can be made to point to objects of different classes (related by inheritance).

Example)

```
class Base
{
    public:
    int b;
    void fun() {};
};
class Derived: public Base
{
    int d;
    void fun() {};
};

Base b_obj;
Derived d_obj;
Base *ptr1 = &b_obj; // Valid => Base class pointer pointing to base class object
Base *ptr2 = &d_obj; // Valid => Base class pointer pointing to derived class object
Derived *ptr3 = &b_obj; // Invalid => Derived class pointer pointing to base class object.
Derived *ptr4 = &d_obj; // Valid => Derived class pointer pointing to derived class object.
```

Q) Why base class pointer can point to derived class object and not vice-versa

R) A derived object is also a base class object (as it's a sub class and thus know members of parent or base class), so it can be pointed to by a base class pointer. However, a base class object is not a derived class object (as it's a super class and don't know members of derived class) so it can't be assigned to a derived class pointer.
Example) Consider a base class Animal and child class Tiger. We can say that Tiger **is a**n animal (*is a relationship*) but we cannot say that animal will be tiger only. Thus, a derived class includes everything that is in the base class. But a base class does not include everything that is in the derived class.

Now, we will consider the case of Base class pointer base class object only i.e. *Base *ptr2 = &d_obj;* But there are some restrictions in using this kind of pointer. We can access only publicly inherited data members/member functions of base class and not the members that originally belong to the derived class.

We cannot access data member '*d*' of derived class using base class pointer and also overrided function '*fun()*' (due to early binding).

Even if a function is overrided (with same name,return type and argument list) then also, base class' member function is called (due to early binding, though Overrided function can be called by late binding if base class overriden function is declared as virtual).

To access members of derived class directly, we either need to create pointer of derived class type only, or they can be accessed using typecasting base class(Base*) pointer to derived class (Derived*). Like *((Derived*)ptr2)->x OR ptr4->x;*

Smart Pointers

Smart Pointer Introduction In C++

The problem with heap memory is that when you don't need it you must deallocate itself yourself. If any programmer forget to write code for deallocation of objects , it causes severe problem like ***memory leak*** which can cause the program to crash.

The languages like Java, C# provide a ***garbage collection mechanism*** to deallocate the object which is not in use.

There was no garbage collection mechanism in C++ because: Garbage collection requires data structures for tracking allocations and/or reference counting. These create overhead in memory, performance, and the complexity of the language. C++ is designed for higher performance side of the tradeoff vs convenience features.

C++ 11 introduces ***smart pointers*** that automatically manage memory and they will deallocate the object when they are not in use i.e. when the pointer is going out of scope automatically it'll automaticaly deallocate the memory. Thus it's fundamental use is to avoid chances of memory leak.

A smart pointer is a **wrapper class (template or generic) over a pointer** (with ***destructor*** to delete objects and free the heap memory and ***operators*** like * and -> ***overloaded***), to manage the lifetime of the pointer.

Since the destructor is automatically called when an object goes out of scope, the dynamically allocated memory would automatically be deleted (or reference count can be decremented).

The objects of smart pointer class look like a pointer but can do many things that a normal pointer can't like automatic destruction (we don't have to explicitly call delete), reference counting and more.

Note: Smart pointers are also useful in the management of resources, such as file handles or network sockets.

Defining our own Smart Pointer

```
template <class T>
class SmartPtr {
```

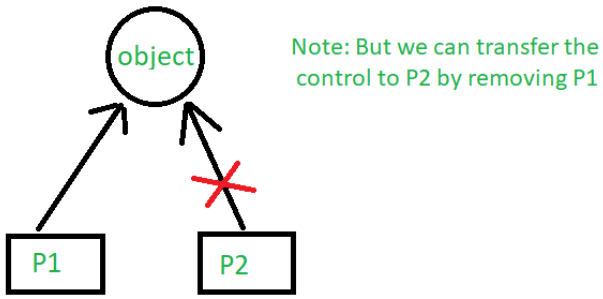
```
    T* ptr; // Actual pointer
public:
    // Explicit(To avoid implicit conversion) Constructor:
    explicit SmartPtr(T* p = NULL) { ptr = p; }

    // Destructor to deallocate objects and free the heap memory
    ~SmartPtr() { delete (ptr); }

    // Overloading dereferencing operator
    T& operator*() { return *ptr; }
};

int main()
{
    SmartPtr<int> ptr(new int());
    *ptr = 20;
    cout << *ptr;

    // We don't need to call delete ptr: when the object ptr goes out of scope, the destructor for it is
    // automatically called and destructor does delete ptr.

    return 0;
}
```

Types of Smart Pointers

1. unique_ptr

Unique Pointer In C++ | Smart Pointer In C++

If you are using a unique pointer then if one object is created and pointer P1 is pointing to this one them only one pointer can point this one at one time. So we can't share with another pointer, but we can transfer the control to P2 by removing P1.



Note: But we can transfer the control to P2 by removing P1

```
#include <iostream>
using namespace std;
#include <memory>
```

```
class Rectangle {
    int length;
    int breadth;

public:
    Rectangle(int l, int b)
    {
        length = l;
        breadth = b;
    }

    int area()
    {
        return length * breadth;
    }
};

int main()
{

    unique_ptr<Rectangle> P1(new Rectangle(10, 5));
    cout << P1->area() << endl; // This'll print 50

    // unique_ptr<Rectangle> P2(P1);

    unique_ptr<Rectangle> P2;
    P2 = move(P1);

    // This'll print 50
    cout << P2->area() << endl;

    // cout<<P1->area()<<endl;
    return 0;
}
```

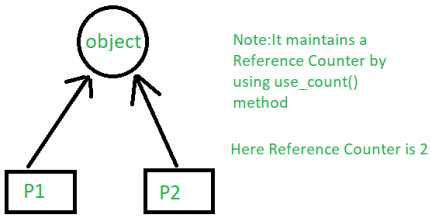
Output:

```
50
50
```

2. shared pointer

Shared Pointer In C++ | Smart Pointer In C++

If you are using shared_ptr then more than one pointer can point to this one object at a time and it'll maintain a ***Reference Counter*** using **use_count() method**.




```
#include <iostream>
using namespace std;
#include <memory>

class Rectangle {
    int length;
    int breadth;

public:
    Rectangle(int l, int b)
    {
        length = l;
        breadth = b;
    }

    int area()
    {
        return length * breadth;
    }
};

int main()
{

    shared_ptr<Rectangle> P1(new Rectangle(10, 5));
    // This'll print 50
    cout << P1->area() << endl;

    shared_ptr<Rectangle> P2;
    P2 = P1;

    // This'll print 50
    cout << P2->area() << endl;

    // This'll now not give an error,
    cout << P1->area() << endl;

    // This'll also print 50 now
    // This'll print 2 as Reference Counter is 2
    cout << P1.use_count() << endl;
    return 0;
}
```

Output:

```
50
50
50
2
```

3. weak pointer

Weak Pointer In C++ | Smart Pointer In C++

It's much more similar to shared_ptr except it'll not maintain a ***Reference Counter***.In this case, a pointer will not have a strong hold on the object. The reason is if suppose pointers are holding the object and requesting for other objects then they may form a ***Deadlock***.