CODEMATES

C++ CHEATSHEET

LITERALS:-

C++ Constants/Literals- Constants refer to fixed values that the program may not alter and they are called literals. Constants can be of any of the basic data types and can be divided into Integer Numerals, Floating-Point Numerals, Characters, Strings and Boolean Values.

```
2125, 0377, 0xff                // Integers (decimal, octal, hex)
3543.0, 1.13e2                  // double (real) numbers
'a', '\191', '\x71'             // Character (literal, octal, hex)
true, false                     // bool constants 1 and 0
'\n', '\\', '\'', '\"'          // Newline, backslash, single quote, double quote
"strings\n"                     // Array of characters ending with newline and \0
 "code" "mates"                 // Concatenated strings
2147483647L, 0x7fffffffl        // Long (32-bit) integers
```

VARIABLE DECLARATIONS AND INITIALIZATIONS:-

A variable provides us with named storage that our programs can manipulate. Each variable in C++ has a specific type, which determines the size and layout of the variable's memory; the range of values that can be stored within that memory; and the set of operations that can be applied to the variable.

```
int x;                  // Declare x to be an integer (value undefined)
int x=255;              // Declare and initialize x to 255
```

```
 short s; long l;          // Usually 16 or 32 bit integer (int may be either)
 char c='a';               // Usually 8 bit character
 unsigned char u=255; signed char s=-1;   // char might be either
unsigned long x=0xffffffffL;           // short, int, long are signed
float f; double d;        // Single or double precision real (never unsigned)
bool b=true;              // true or false, may also use int (1 or 0)
int a, b, c;              // Multiple declarations
int a[10];                // Array of 10 integers (a[0] through a[9])
int a[]={0,1,2};          // Initialized array (or a[3]={0,1,2}; )
int a[2][3]={{1,2,3},{4,5,6}};  // Array of array of integers
char s[]="hello";         // String (6 elements including '\0')
int* p;                   // p is a pointer to (address of)
int char* s="hello";          // s points to unnamed array containing "hello"
void* p=NULL;             // Address of untyped memory (NULL is 0)
int& r=x;                 // r is a reference to (alias of) int x
enum weekend {SAT,SUN};   // weekend is a type with values SAT and SUN
 enum weekend day;        // day is a variable of type weekend
 enum weekend {SAT=0,SUN=1};  // Explicit representation as int
 enum {SAT,SUN} day;      // Anonymous enum
typedef String char*;     // String s; means char* s;
const int c=3;            // Constants must be initialized, cannot assign to const
 int* p=a;           // Contents of p (elements of a) are constant
 int* const p=a;           // p (but not contents) are constant const
int* const p=a;     // Both p and its contents are constant const
int& qw=x;          // qw cannot be assigned to change x
```

STATEMENTS:- **Statements are fragments of the C++ program that are executed in sequence. The body of any function is a sequence of statements. C++ includes the following types of statements:**

1) expression statements;
2) compound statements;
3) selection statements;
4) iteration statements;
5) jump statements;
6) declaration statements;
7) try blocks;
8) atomic and synchronized blocks

```
x=y;                    // Every expression is a statement

int x;                  // Declarations are statements

 ;                      // Empty statement

{                       // A block is a single statement

int x;                  // Scope of x is from declaration to end of block

 a;                     // In C, declarations must precede statements

}

if (x) a;               // If x is true (not 0), evaluate a

else if (y) b;          // If not x and y (optional, may be repeated)

else c;                 // If not x and not y (optional)

while (x) a;            // Repeat 0 or more times while x is true

 for (x; y; z) a;       // Equivalent to: x; while(y) {a; z;}

do a; while (x);        // Equivalent to: a; while(x) a;

switch (x) {            // x must be int

case X1: a;             // If x == X1 (must be a const), jump here

case X2: b;             // Else if x == X2, jump here

default: c;             // Else jump here (optional) }

break;                  // Jump out of while, do, or for loop, or switch

continue;               // Jump to bottom of while, do, or for loop

return x;               // Return x from function to caller

 try { a; }

catch (T t) { b; }      // If a throws a T, then jump here

catch (...) { c; }      // If a throws something else, jump h
```

EXPRESSIONS:- C++ expression consists of operators, constants, and variables which are arranged according to the rules of the language. It can also contain function calls which return values.

Operators are grouped by precedence, highest first. Unary operators and assignment evaluate right to left. All others are left to right. Precedence does not affect order of evaluation, which is undefined. There are no run time checks for arrays out of bounds, invalid pointers, etc.

```
x * y                 // Multiply
 x / y                // Divide (integers round toward 0)
 x % y                // Modulo (result has sign of x)
 x + y                // Add, or &x[y]
x - y                 // Subtract, or number of elements from *x to *y
++x                   // Add 1 to x, evaluates to new value (prefix)
--x                   // Subtract 1 from x, evaluates to new value
 ~x                   // Bitwise complement of x
 !x                   // true if x is 0, else false (1 or 0 in C)
 -x                   // Unary minus
+x                    // Unary plus (default)
&x                    // Address of x
T::X                  // Name X defined in class T
N::X                  // Name X defined in namespace
 N ::X                // Global name X
 t.x                  // Member x of struct or class t
p->x                  // Member x of struct or class pointed to by p
a[i]                  // i'th element of array a
f(x,y)                // Call to function f with arguments x and y
T(x,y)                // Object of class T initialized with x and y
x++                   // Add 1 to x, evaluates to original x (postfix)
x--                   // Subtract 1 from x, evaluates to original x
typeid(x)             // Type of x
typeid(T)             // Equals typeid(x) if x is a T
dynamic_cast<T>(x)    // Converts x to a T, checked at run time
```

```
 static_cast<T>(x)        // Converts x to a T, not checked
 reinterpret_cast<T>(x)   // Interpret bits of x as a T
const_cast<T>(x)          // Converts x to same type T but not const
sizeof x                  // Number of bytes used to represent object x
sizeof(T)                 // Number of bytes to represent type T
*p                        // Contents of address p (*&x equals x)
new T                     // Address of newly allocated T object
new T(x, y)               // Address of a T initialized with x, y
 new T[x]                 // Address of allocated n-element array of T
delete p                  // Destroy and free object at address p
delete[] p                // Destroy and free array of objects at p
(T) x                     // Convert x to T (obsolete, use .._cast<T>(x))
 x << y                   // x shifted y bits to left (x * pow(2, y))
x >> y                    // x shifted y bits to right (x / pow(2, y))
x < y                     // Less than
x <= y                    // Less than or equal to
 x > y                    // Greater than
x >= y                    // Greater than or equal to
x == y                    // Equals
x != y                    // Not equals
x & y                     // Bitwise and (3 & 6 is 2)
x ^ y                     // Bitwise exclusive or (3 ^ 6 is 5)
x | y                     // Bitwise or (3 | 6 is 7)
x && y                    // x and then y (evaluates y only if x (not 0))
x || y                    // x or else y (evaluates y only if x is false (0))
x = y                     // Assign y to x, returns new value of x
x += y                    // x = x + y, also -= *= /= <<= >>= &= |= ^=
x ? y : z                 // y if x is true (nonzero), else z
throw x                   // Throw exception, aborts if not caught
x , y                     // evaluates x and y, returns y (seldom used)
```

FUNCTIONS:-

A function is a block of code which only runs when it is called. You can pass data, known as parameters, into a function. Functions are used to perform certain actions, and they are important for reusing code: Define the code once, and use it many times.

Function parameters and return values may be of any type. A function must either be declared or defined before it is used. It may be declared first and defined later. Every program consists of a set of a set of global variable declarations and a set of function definitions (possibly in separate files), one of which must be:

int main() { statements... } or int main(int argc, char* argv[]) { statements... }

argv is an array of argc strings from the command line. By convention, main returns status 0 if successful, 1 or higher for errors.

Functions with different parameters may have the same name (overloading). Operators except :: . .* ?: may be overloaded. Precedence order is not affected. New operators may not be created.

int f(int x, int); // f is a function taking 2 integers and returning

int void f(); // f is a procedure taking no arguments

void f(int a=0); // f() is equivalent to f(0)

f(); // Default return type is int

inline f(); // Optimize for speed

f() { statements; } // Function definition (must be global)

T operator+(T x, T y); // a+b (if type T) calls operator+(a, b)

T operator-(T x); // -a calls function operator-(a)

T operator++(int); // postfix ++ or -- (parameter ignored)

extern "C" {void f();} // f() was compiled in C

CLASSES:- A class in C++ is the building block, that leads to Object-Oriented programming. It is a user-defined data type, which holds its own data members and member functions, which can be accessed and used by creating an instance of that class. A C++ class is like a blueprint for an object. For Example: Consider the Class of Cars. There may be many cars with different names and brand but all of them will share some common properties like all of them will have *4 wheels*, *Speed Limit*, *Mileage range* etc. So here, Car is the class and wheels, speed limits, mileage are their properties.

- A Class is a user defined data-type which has data members and member functions.
- Data members are the data variables and member functions are the functions used to manipulate these variables and together these data members and member functions defines the properties and behaviour of the objects in a Class.
- In the above example of class *Car*, the data member will be *speed limit*, *mileage* etc and member functions can be *apply brakes*, *increase speed* etc.

```
class T {               // A new type
private:                // Section accessible only to T's member
functions protected:            // Also accessible to classes derived from T
public:                 // Accessible to all
int x;                  // Member data
void f();               // Member function
void g() {return;}      // Inline member function
void h() const;         // Does not modify any data members
int operator+(int y);   // t+y means t.operator+(y)
int operator-();        // -t means t.operator-()   T
(): x(1) {}             // Constructor with initialization list
T(const T& t): x(t.x) {}  // Copy constructor
T& operator=(const T& t) {x=t.x; return *this; }  // Assignment operator
 ~T();                  // Destructor (automatic clean-up routine)
explicit T(int a);      // Allow t=T(3) but not t=3
operator int() const {return x;}  // Allows int(t)
friend void i();        // Global function i() has private access
friend class U;         // Members of class U have private access
static int y;           // Data shared by all T objects
 static void l();       // Shared code.  May access y but not x
class Z {};             // Nested class T::Z
```

```
 typedef int V;          // T::V means int };
void T::f() {            // Code for member function f of class T
this->x = x;}           // this is address of self (means x=x;)
 int T::y = 2;           // Initialization of static member (required)
 T::l();                // Call to static member
 struct T {             // Equivalent to: class T { public:
virtual void f();       // May be overridden at run time by derived class
virtual void g()=0; };  // Must be overridden (pure virtual)
class U: public T {};     // Derived class U inherits all members of base T
class V: private T {};    // Inherited members of T become private
 class W: public T, public U {};  // Multiple inheritance
class X: public virtual T {}; // Classes derived from X have base T
```

TEMPLATES:- Templates are powerful features of C++ which allows you to write generic programs. In simple terms, you can create a single function or a class to work with different data types using templates. Templates are often used in larger codebase for the purpose of code reusability and flexibility of the programs.

```
template <class T> T f(T t);        // Overload f for all types
 template <class T> class X {       // Class with type parameter T
 X(T t); };                         // A constructor
template <class T> X<T>::X(T t) {}  // Definition of constructor
X<int> x(3);                        // An object of type "X of int"
template <class T, class U=T, int n=0>  // Template with default parameters
```

C/C++ STANDARD LIBRARY:-

The Standard Template Library (STL) is a set of C++ template classes to provide common programming data structures and functions such as lists, stacks, arrays, etc. It is a library of container classes, algorithms, and iterators. It is a generalized library and so, its components are parameterized. A working knowledge of template classes is a prerequisite for working with STL.

STL has four components:-

- Algorithms
- Containers
- Functions
- Iterators

SOME IMPORTANT AND MOST USED STL FUNCTIONS

STRING - Variable sized character array

```
string s1, s2="hello";    // Create strings
 s1.size(), s2.size();     // Number of characters: 0, 5
s1 += s2 + ' ' + "world"; // Concatenation
s1 == "hello world"       // Comparison, also <, >, !=, etc.
s1[0];                    // 'h'
 s1.substr(m, n);         // Substring of size n starting at s1[m]
s1.c_str();               // Convert to const char*
getline(cin, s);          // Read line ending in '\n'
```

VECTOR- Variable sized array/stack with built in memory allocation

```
vector<int> a(10);        // a[0]..a[9] are int (default size is 0)
a.size();                 // Number of elements (10)
a.push_back(3);           // Increase size to 11, a[10]=3
 a.back()=4;              // a[10]=4;
a.pop_back();             // Decrease size by 1
```

```
a.front();              // a[0];
a[20]=1;                // Crash: not bounds checked
a.at(20)=1;             // Like a[20] but throws out_of_range()
 for (vector<int>::iterator p=a.begin(); p!=a.end(); ++p)
*p=0;                   // Set all elements of a to 0
vector<int> b(a.begin(), a.end());  // b is copy of a
vector<T> c(n, x);      // c[0]..c[n-1] initialize to x T d[10];
vector<T> e(d, d+10);   // e is initialized from d
```

DEQUE - Double ended queue

deque (usually pronounced like *"deck"*) is an irregular acronym of double-ended queue. Double-ended queues are sequence containers with dynamic sizes that can be expanded or contracted on both ends (either its front or its back).

Specific libraries may implement *deques* in different ways, generally as some form of dynamic array. But in any case, they allow for the individual elements to be accessed directly through random access iterators, with storage handled automatically by expanding and contracting the container as needed.

deque<T> is like vector<T>, but also supports:

```
a.push_front(x);        // Puts x at a[0], shifts elements toward back
a.pop_front();          // Removes a[0], shifts toward front
```

UTILITY (Pair) :-

```
pair<string, int> a("hello", 3);  // A 2-element struct
a.first;                // "hello"
a.second;               // 3
```

MAP (associative array):- Maps are associative containers that store elements formed by a combination of a *key value* and a *mapped value*, following a specific order.

```
map<string, int> a;       // Map from string to int
a["hello"]=3;             // Add or replace element a["hello"]
for (map<string, int>::iterator p=a.begin(); p!=a.end(); ++p)
cout << (*p).first << (*p).second;  // Prints hello, 3
 a.size();                // 1
```

ALGORITHM:- The algorithms library defines functions for a variety of purposes (e.g. searching, sorting, counting, manipulating) that operate on ranges of elements. Note that a range is defined as [first, last) where last refers to the element *past* the last element to inspect or modify.

Non-modifying sequence operations

Defined in header `<algorithm>`

all_ofany_ofnone_of (C++11)(C++11)(C++11)	checks if a predicate is true for all, any or none of the elements i (function template)
for_each	applies a function to a range of elements (function template)
for_each_n (C++17)	applies a function object to the first n elements of a sequence (function template)
countcount_if	returns the number of elements satisfying specific criteria (function template)
mismatch	finds the first position where two ranges differ (function template)
findfind_iffind_if_not (C++11)	finds the first element satisfying specific criteria (function template)
find_end	finds the last sequence of elements in a certain range (function template)
find_first_of	searches for any one of a set of elements (function template)

adjacent_find	finds the first two adjacent items that are equal (or satisfy a given (function template)
search	searches for a range of elements (function template)
search_n	searches a range for a number of consecutive copies of an elemen (function template)

Modifying sequence operations

Defined in header <algorithm>

copycopy_if (C++11)	copies a range of elements to a new location (function template)
copy_n (C++11)	copies a number of elements to a new location (function template)
copy_backward	copies a range of elements in backwards order (function template)
move (C++11)	moves a range of elements to a new location (function template)
move_backward (C++11)	moves a range of elements to a new location in backwards order (function template)
fill	copy-assigns the given value to every element in a range (function template)
fill_n	copy-assigns the given value to N elements in a range (function template)
transform	applies a function to a range of elements, storing results in a dest (function template)
generate	assigns the results of successive function calls to every element i (function template)
generate_n	assigns the results of successive function calls to N elements in a (function template)
removeremove_if	removes elements satisfying specific criteria (function template)

remove_copyremove_copy_if	copies a range of elements omitting those that satisfy specific cri (function template)
replacereplace_if	replaces all values satisfying specific criteria with another value (function template)
replace_copyreplace_copy_if	copies a range, replacing elements satisfying specific criteria wit (function template)
swap	swaps the values of two objects (function template)
swap_ranges	swaps two ranges of elements (function template)
iter_swap	swaps the elements pointed to by two iterators (function template)
reverse	reverses the order of elements in a range (function template)
reverse_copy	creates a copy of a range that is reversed (function template)
rotate	rotates the order of elements in a range (function template)
rotate copy	copies and rotate a range of elements (function template)
shift_leftshift_right (C++20)	shifts elements in a range (function template)
random_shuffleshuffle (until C++17)(C++11)	randomly re-orders elements in a range (function template)
sample (C++17)	selects n random elements from a sequence (function template)
unique	removes consecutive duplicate elements in a range (function template)
unique_copy	creates a copy of some range of elements that contains no consec (function template)

Partitioning operations

Defined in header <algorithm>

is_partitioned (C++11)	determines if the range is partitioned by the given predicate (function template)
partition	divides a range of elements into two groups (function template)
partition_copy (C++11)	copies a range dividing the elements into two groups (function template)
stable_partition	divides elements into two groups while preserving their relative (function template)
partition_point (C++11)	locates the partition point of a partitioned range (function template)

Sorting operations

Defined in header <algorithm>

is_sorted (C++11)	checks whether a range is sorted into ascending order (function template)
is_sorted_until (C++11)	finds the largest sorted subrange (function template)
sort	sorts a range into ascending order (function template)
partial_sort	sorts the first N elements of a range (function template)
partial_sort_copy	copies and partially sorts a range of elements (function template)
stable_sort	sorts a range of elements while preserving order between equal e (function template)
nth_element	partially sorts the given range making sure that it is partitioned b (function template)

Binary search operations (on sorted ranges)

Defined in header <algorithm>

lower_bound	returns an iterator to the first element *not less* than the given valu (function template)
upper_bound	returns an iterator to the first element *greater* than a certain valu (function template)
binary_search	determines if an element exists in a certain range (function template)
equal_range	returns range of elements matching a specific key (function template)

Other operations on sorted ranges

Defined in header <algorithm>

merge	merges two sorted ranges (function template)
inplace_merge	merges two ordered ranges in-place (function template)

Set operations (on sorted ranges)

Defined in header <algorithm>

includes	returns true if one set is a subset of another (function template)
set_difference	computes the difference between two sets (function template)
set_intersection	computes the intersection of two sets (function template)
set_symmetric_difference	computes the symmetric difference between two sets (function template)
set_union	computes the union of two sets (function template)

Heap operations

Defined in header <algorithm>

is_heap (C++11)	checks if the given range is a max heap (function template)
is_heap_until (C++11)	finds the largest subrange that is a max heap (function template)
make_heap	creates a max heap out of a range of elements (function template)
push_heap	adds an element to a max heap (function template)
pop_heap	removes the largest element from a max heap (function template)
sort_heap	turns a max heap into a range of elements sorted in ascending or (function template)

Minimum/maximum operations

Defined in header <algorithm>

max	returns the greater of the given values (function template)
max_element	returns the largest element in a range (function template)
min	returns the smaller of the given values (function template)
min_element	returns the smallest element in a range (function template)
minmax (C++11)	returns the smaller and larger of two elements (function template)
minmax_element (C++11)	returns the smallest and the largest elements in a range (function template)
clamp (C++17)	clamps a value between a pair of boundary values (function template)

Comparison operations

Defined in header <algorithm>

equal	determines if two sets of elements are the same (function template)
lexicographical_compare	returns true if one range is lexicographically less than another (function template)
lexicographical_compare_three_way (C++20)	compares two ranges using three-way comparison (function template)

Permutation operations

Defined in header <algorithm>

is_permutation (C++11)	determines if a sequence is a permutation of another sequence (function template)
next_permutation	generates the next greater lexicographic permutation of a range o (function template)
prev_permutation	generates the next smaller lexicographic permutation of a range o (function template)

Numeric operations

Defined in header <numeric>

iota (C++11)	fills a range with successive increments of the starting value (function template)
accumulate	sums up a range of elements (function template)
inner_product	computes the inner product of two ranges of elements (function template)
adjacent_difference	computes the differences between adjacent elements in a range (function template)
partial_sum	computes the partial sum of a range of elements (function template)
reduce	similar to std::accumulate, except out of order (function template)

(C++17)	
exclusive_scan (C++17)	similar to std::partial_sum, excludes the ith input element from th (function template)
inclusive_scan (C++17)	similar to std::partial_sum, includes the ith input element in the i (function template)
transform_reduce (C++17)	applies an invocable, then reduces out of order (function template)
transform_exclusive_scan (C++17)	applies an invocable, then calculates exclusive scan (function template)
transform_inclusive_scan (C++17)	applies an invocable, then calculates inclusive scan (function template)
Operations on uninitialized memory Defined in header <memory>	
uninitialized_copy	copies a range of objects to an uninitialized area of memory (function template)
uninitialized_copy_n (C++11)	copies a number of objects to an uninitialized area of memory (function template)
uninitialized_fill	copies an object to an uninitialized area of memory, defined by a (function template)
uninitialized_fill_n	copies an object to an uninitialized area of memory, defined by a (function template)
uninitialized_move (C++17)	moves a range of objects to an uninitialized area of memory (function template)
uninitialized_move_n (C++17)	moves a number of objects to an uninitialized area of memory (function template)
uninitialized_default_construct (C++17)	constructs objects by default-initialization in an uninitialized area (function template)

uninitialized_default_construct_n (C++17)	constructs objects by default-initialization in an uninitialized area (function template)
uninitialized_value_construct (C++17)	constructs objects by value-initialization in an uninitialized area (function template)
uninitialized_value_construct_n (C++17)	constructs objects by value-initialization in an uninitialized area (function template)
destroy_at (C++17)	destroys an object at a given address (function template)
destroy (C++17)	destroys a range of objects (function template)
destroy_n (C++17)	destroys a number of objects in a range

STDLIB.H, CSTDLIB (Misc. functions):-

```
atof(s); atol(s); atoi(s);// Convert char* s to float, long, int
rand(), srand(seed);      // Random int 0 to RAND_MAX, reset rand()
void* p = malloc(n);      // Allocate n bytes.  Obsolete: use new
free(p);                  // Free memory.  Obsolete: use delete
exit(n);                  // Kill program, return status n
system(s);                // Execute OS command s (system dependent)
getenv("PATH");           // Environment variable or 0 (system dependent)
abs(n); labs(ln);         // Absolute value as int, long
```

STRING.H, CSTRING - Character array handling functions)

Strings are type char[] with a '\0' in the last element used.

strcpy(dst, src); // Copy string. Not bounds checked

strcat(dst, src); // Concatenate to dst. Not bounds checked

strcmp(s1, s2); // Compare, <0 if s1<s2, 0 if s1==s2, >0 if s1>s2

strncpy(dst, src, n); // Copy up to n chars, also strncat(), strncmp()

strlen(s); // Length of s not counting \0 strchr(s,c);

strrchr(s,c);// Address of first/last char c in s or 0

strstr(s, sub); // Address of first substring in s or 0 // mem... functions are for any pointer types (void*), length n bytes

memmove(dst, src, n); // Copy n bytes from src to dst

memcmp(s1, s2, n); // Compare n bytes as in strcmp

memchr(s, c, n); // Find first byte c in s, return address or 0

MATH.H, CMATH -Floating point math

sin(x); cos(x); tan(x); // Trig functions, x (double) is in radians

asin(x); acos(x); atan(x);// Inverses

atan2(y, x); // atan(y/x)

sinh(x); cosh(x); tanh(x);// Hyperbolic exp(x);

log(x); log10(x); // e to the x, log base e, log base 10

pow(x, y); sqrt(x); // x to the y, square root

ceil(x); floor(x); // Round up or down (as a double)

fabs(x); fmod(x, y); // Absolute value, x mod y

TIME.H, CTIME -Clock

clock()/CLOCKS_PER_SEC; // Time in seconds since program started

time_t t=time(0); // Absolute time in seconds or -1 if unknown

tm* p=gmtime(&t); // 0 if UCT unavailable, else p->tm_X where X is:

sec, min, hour, mday, mon (0-11), year (-1900), wday, yday, isdst

asctime(p); // "Day Mon dd hh:mm:ss yyyy\n"

asctime(localtime(&t)); // Same format, local time

IOSTREAM.H, IOSTREAM (Replaces stdio.h):-

cin >> x >> y; // Read words x and y (any type) from stdin

cout << "x=" << 3 << endl; // Write line to stdout

cerr << x << y << flush; // Write to stderr and flush

c = cin.get(); // c = getchar();

cin.get(c); // Read char cin.getline(s, n, '\n'); // Read line into char s[n] to '\n' (default)

if (cin) // Good state (not EOF)? // To read/write any type T:

istream& operator>>(istream& i, T& x) {i >> ...; x=...; return i;}

ostream& operator<<(ostream& o, const T& x) {return o << ...;}

FSTREAM.H, FSTREAM (File I/O works like cin, cout as above) :-

ifstream f1("filename"); // Open text file for reading

if (f1) // Test if open and input available

f1 >> x; // Read object from file

f1.get(s); // Read char or line

f1.getline(s, n); // Read line into string s[n]

ofstream f2("filename"); // Open file for writing

PREPROCESSOR:-

C++ Preprocessor. The preprocessors are the directives, which give instructions to the compiler to preprocess the information before actual compilation starts. All preprocessor directives begin with #, and only white-space characters may appear before a preprocessor directive on a line.

```
// Comment to end of line
/* Multi-line comment */
#include <stdio.h>        // Insert standard header file
#include "myfile.h"       // Insert file in current directory
#define X some text       // Replace X with some text
#define F(a,b) a+b        // Replace F(1,2) with 1+2
#define X \   some text             // Line continuation
#undid X                  // Remove definition
#if defined(X)            // Condional compilation (#ifdef X)
#else                     // Optional (#ifndef X or #if !defined(X))
#endif                    // Required after #if, #ifdef
```

STORAGE CLASSES:-

Storage class specifiers supported in C++ are auto, register, static, extern and mutable. This is the default storage class we have been using so far. It applies to local variables only and the variable is visible only inside the function in which it is declared and it dies as soon as the function execution is over.

```
int x;                    // Auto (memory exists only while in scope)
static int x;             // Global lifetime even if local scope
extern int x;             // Information only, declared elsewhere
```