OOP : — encapsulate data (attributes) and function (behavior) into packages "class"

classes have the property of information hiding (not allowed to know how other classes are implemented = implementation details are hidden within the classes themselves) [although ~~the~~ class objects may know how to communicate with one another across well-defined interfaces.]

class ≡ user-defined type ≡ programmer-defined type
(contains data + set of functions that manipulate the data)

(data members) attributes — (member function / method) behaviors/operations

the dot operator accesses a structure or class member via the variable name for the object or via a reference to the object.

→ ✓ Object . member ✓ reference . member

the arrow operator accesses a structure member or class member via a pointer to an object.

✓ Pointer → member ✓

✓ *Pointer . member

Implementing abstract data type :—

public member functions (also called public services, public behaviors or interface of the class) are used by clients (portions of a program that are users). (in .h file)

Private members are accessible only to member functions and ~~[illegible]~~ friends of the class.

the data representation used within the class is of no concern to the class's clients (~~[illegible]~~ implementation of a class is said to be hidden from its clients)

⊛ initialize data member by constructor
⊛ assign value to date member by set function
⊛ Clients should have access to a class's interface, but should not have access to a class's implementation.
⊛ ~~separate~~ separates the interface of a class from its implementation by declaring member functions inside class definition (via their function prototypes) and defining those member functions outside that class definition. (suggestion)
⊛ Member functions defined outside a class definition can be inlined by explicitly using keyword inline.
⊛ Physically, objects contain only data and share only one common copy of member functions among themselves.
⊛ hidden global variable can be accessed with (::)
⊛ class definition is normally placed in header file (.h), enclosed with preprocessor code

time1.h
```
# ifndef TIME1_H
# define TIME1_H
   class Time { - - - .
         - - .
   };
# endif
```

⊛ member-function definition is normally placed in source-code file (.cpp) with the inclusion of programmer-defined header files ("time1.h").

time1.cpp
```
#include <iostream>
#include "time1.h"
   - - -
Time::Time() { - - -   }
void Time::setTime(int h, int m, int s) { - - - }
  - - - -
}
```

```
main.cpp  { #include <iostream>
            #include "time1.h"
            int main() { - - - -
            }
```

- the default access mode for classes is private but can be set to public, private, protected.
- " " " " structure is public but can be set to public, private, protected.
- provide set function in public member of class to set the value of private data member and
- " get function " " " " " to get " " " " ".
- provide a constructor to ensure that every object is properly initialized.
- ~~constructor are like~~
- default constructor is created by specifying default arguments and can be invoked with no arguments.
- there can be only one default constructor per class.
- ~~def~~ declare default function argument values only in the function prototype within the class definition in the header file (suggested)
- generally, destructor calls are made in the reverse order of corresponding constructor calls except in case of storage classes of objects
- the constructor for a static local object is called only once when execution first reaches the point ~~of~~ where the object is defined - ~~[illegible]~~ corresponding destructor is called when main terminates or the program calls function exit.
- Global and static objects are destroyed in the reverse order of their creation.

⊛ set and get functions need not be called set and get specifically.
⊛ classes often provide public member functions to allow clients of the class to set (write) or get (read) the values of private data members.
⊛ each get function simply returns the appropriate data member's value.
⊛ Returning pointers or references to private data is dangerous.
(Never have a public member function return a non-const reference (or pointer) to a private data member.)

⊛ ~~pass an object~~
⊛ try to pass an object as pass-by-const-reference
⊛ a function is specified as const both in its prototype and in its definition by inserting the keyword const after the function's parameter list and, in the case of the function definition, before the left brace that begins the function body.

```
        int getHour () const;
        int Time :: getHour () const { - - -
                                       - - -
                                    }
```

⊛ defining as const a member function that modifies a data member of an object is a compiler error.
⊛ defining as const a member function that calls a non-const member function of the class on the same instance of the class is a compiler error.
⊛ invoking a non-const member function on a const object is compiler error.

⊛ constant data members (const objects and const "variables") and data members declared as references must be initialized with member ~~[illegible]~~ initializer syntax.

$$\text{constructor name (parameter list)}: \underbrace{\text{datamembername1(value1), datamember2(value2), ...}}_{\text{member initializers syntax}} \{ - - - \}$$

⊛ initialize member objects explicitly through member initializers.

~~member const object~~ ≡ ~~object created by different other class but~~ ~~but~~

member object ≡ object of a class (A) as its member but the object is created by other class (B).

(a class has object of other class as member)

⊛ friend class myclass; // to declare all member functions as friends
name

friend void setX(Count &, int); // standalone function of other class as friend

⊛ friends are not member functions.

⊛ friends declarations can be placed anywhere in a class (no member access specifier)

⊛ friendship is not transitive. $A \not\to B \not\to C$ / $A \to B$, $B \to C$

⊛ friendship is not symmetric. $A \to B$ / $B \not\to A$

⊛ a friend function of a class ~~is~~ has the right to access/modify all members (including non-public members) of the class.
private

⊛ every object has access to its own address through a pointer called this.

⊛ this pointer is used to enable cascaded member-function calls (multiple functions are invoked in the same statement)

[illegible]: object . member-function1 . member-function2 . member-function3. - - - -
1, 2, 3 → (left to right)

each member-function returns a reference to the object as (*this)

年 月 日 第 页 9156051

cout << . . . << - - . << . . .

⊛ for any built-in or user-defined type, operator <u>new</u> and <u>delete</u> are used to allocate or deallocate memory.
⊛ should include standard header <new>
⊛ operator <u>new</u> creates an <u>object</u> of the proper size for <u>specified type</u>.

```
Time *timePtr;
        timePtr = new Time;
```

✓
⊛ an <u>initializer</u>/argument list may be provided for a newly created object:

```
double *ptr = new double(3.1415);
Time *timePtr = new Time(12, 0, 0);
```

✓✓
⊛ [new creates specified type <u>object</u> ~~with~~ and ~~[illegible]~~ initializes the <u>object</u> with the value initializing]

<u>syntax</u>: built-in / user-defined specified type variable = new specified type (initializer (value) / argument list (values))

similarly, delete variable ;
            delete [] ~~variable~~ array ;

Static class member :-

- A static class variable is used, incase, only one copy of a variable should be shared by all objects of a class.
- A static class variable represents "class-wide" information (a property of the class)
- static data members have class scope, although static data members may seem like global variable. (not file scope)
- each static data member must be initialized once (and only once) at file scope (not in the body of class definition).
- donot use the this pointer in a static member function.
- donot declare a static member function const.
- After deleting dynamically allocated memory, set the pointer that referred to that memory to 0. This disconnects the pointer from the previously allocated space on the free store.

date Abstraction & Information hiding :-

* classes normally hide their implementation details from the clients of the classes (information hiding)
* describing the functionality of a class independent of its implementation is called (data abstraction). [illegible]
* C++ classes defined so-called ADT (Abstract Data Type).
* Abstract data types are essentially ways of ~~representing~~ representing real-world notions to some satisfactory level of precision within a computer system.
* An abstract data type actually captures two notions, namely a data representation and the operations that are allowed on those data.

e.g.: Array (ADT)
String (")
Queue (")

Container Class (collection class) and iterators :-

* Container classes or collection classes are designed to hold collections of objects.
they provide services such as insertion, deletion, searching, sorting, testing an item to determine whether it is a member of the collection.

examples of container classes: Array, stack, queue, tree, linked list, ...

• Iterator objects/iterators are object that return the next item of a collection (or perform some action on the next item of a collection).

* Proxy Class = ~~contains only the public interface to your~~
↑ ~~class enables~~

• are ~~created~~ created to hide the implementation details of a class to prevent access to proprietary information (including private data) and proprietary program logic in a class.

• it is done by providing clients of your class with proxy class that knows only the public interface to your class enables the clients to use your class's services without giving the client access to your class's implementation details.

Operator overloading:-

⊛ Operators are ~~[illegible]~~ overloaded by writing a function definition with function name as keyword Operator followed by symbol for the operator being overloaded.

e.g.: operator+
operator−

⊛ The point of operator overloading is ~~a~~ to provide the same concise and familiar expressions for user-defined types that C++ provides with its rich collection of operators for built-in types.

⊛ ~~[illegible]~~ all operators except (. .* :: ?:) can be overloaded.

⊛ operators should be loaded ~~explo~~ explicitly.

⊛ overloading operator = + { object 2 = object 2 + object 1; ⇒
⇒ object 2 += object 1;

⊛ the operator overloading function must be declared as a class member when overloading (), [], → or any of the assignment operators

- when an operator function is implemented as a member function, the leftmost (or only) operand must be an object (or a reference to an object) of the operator's class.

class A { - - - operator>>(___ , ___)

object of class A
reference to

- ~~If the left operand must be an object (or a reference to an object) of the~~
- If the left operand must be an object of a different class or a built-in type, this operator function must be non-member function.

  A non-member operator function needs to be a friend if that function must access private or protected members of that class directly.
- it is possible to overload an operator as a non-member, non-friend function, but such function requiring access to a class's private or protected data would need to use set or get functions provided in that class's public interface.
- static member functions only can access static data members of the class.
- copy constructor must receive its argument pass-by-reference:

Running C++ on cygwin = GNU + Cygnus + Window
(linux-like environment for Windows)

www.cygwin.com

```
$ cd   D:/C++/Project
$dir
 $ g++  Hello.cpp
$ dir
 Hello.cpp   Hello.cpp~  a.exe
$ ./a
 Helloworld!
$ g++ --help
$g++ -o  Hello.exe  Hello.cpp
$ ./Hello
 HelloWorld!
```

www.gnu.org
eclipse

for gtk+ :

```
$ gtk-config --help
Hello.cpp{ #include <gtk/gtk.h>
int main(int argc, char*argv[])
{ GtkWidget * Window;
   - - --  ..
   - . .
   - gtk_main();..
   - ..
}
```

~~gtk~~ Reference manual:→ developer.gnome.org/

```
D:\C++\Project> set  PATH= C:\cygwin\bin
D:\C++\Project> Hello
           HelloWorld!
```
} on windows prompt

Hello.bat { Set PATH= C:\cygwin\bin
Hello } in C++\Project folder

```
D:\C++\Project> Hello
  HelloWorld!
```

— [redacted] (OOAD with ~~design~~ Applications) —

~~[illegible]~~ Abstraction:— ~~simplified description~~

Abstraction denotes the essential characteristics of an object that distinguish it from all other kinds of objects and thus provide crisply defined conceptual boundaries, relative to the ~~[illegible]~~ viewer's perspective.

Encapsulation — is the process of compartmentalizing the elements of an abstraction that constitute its structure and behavior; encapsulation serves to separate the ~~compartment~~ contractual interface of an abstraction and its implementation.

Modularity — is the property of a system that has been decomposed into a set o cohesive and loosely coupled modules.

Hierarchy — is the ranking or ~~[illegible]~~ ordering of abstractions.

Typing — is the enforcement of the class of an object, ~~[illegible]~~ such that objects of different types may not be interchanged, or at the most, they ~~[illegible]~~ may be interchanged only in very restricted ways.

Concurrency — is the property that distinguishes an active object from one that is no active

Persistence — is the property of an object through which its existence transcends time (i.e, the object continues to exist after its creator ceases to exist) a space (i.e, the object's location moves from the address space in which it was created)

* ~~an~~ object has state, ~~[illegible]~~ behavior, and identity; the structure and behavior similar objects are defined in their common class, the terms instance object are interchangeable.

* the state of an object encompasses all of the (usually static) properties of the object the current (usually dynamic) values of each of these properties.

* behavior is how an object acts and reacts, in terms of its state changes and message passing.

* the state of an object represents the cumulative results of its behavior.

* identity is that property of an object which distinguishes it from all other objects.

* class is a set of objects that share a common structure and a common behavior.

Converting between types = C++

* Conversion constructors (single argument constructors) turn objects of ~~[illegible]~~ other types (including built-in types) into objects of a particular class.
* A conversion operator / cast operator can be used to convert an object of one class into an object of another class or into an object of a built-in type.

$$\underbrace{A}_{\text{object of class/user-defined type}} :: \underbrace{\text{operator}}_{\text{keyword for operator overloading or casting}} \quad \underbrace{\text{char*()}}_{\text{Built-in type}} \text{ const;}$$

~~it shows~~ Overloaded cast operator function for converting an object of user-defined type A into a temporary char* object.

A:: operator int() const;

✓ ⇒ ~~[illegible]~~ | one class object :: operator other class object () const;
~~[illegible]~~
or
object of built-in type

* post increment and post decrement operators return objects by value, whereas the preincrement ~~or~~ and predecrement operators returns object by reference.
* the argument 0 is strictly a "dummy value" that enables compiler to distingui between pre- and post (increment/decrement) operator functions.

~~operator ++ (d1, 0)~~
d1. operator ++ (0)

* standard vector class has ~~many~~ many ~~similar~~ same features as ~~class~~ Array.
(vector is a dynamically ~~vector is array~~ resizable array)

* with OOP, programmers focus on the commonalities among objects in the system, rather then on the special cases. This process is called abstraction.

---

* Inheritance =
* C++ offers three kinds of inheritance - public, protected, private
* C++ supports multiple inheritance also.
* friend functions are not inherited.
* "is a" relationship is inheritance. (
* "has a" relationship is composition.
* base class's protected members can be accessed by members and friends of that base class and by members and friends of any classes derived from that base class.
* use protected access specifier when a base class should provide a service (i.e., a member function) only to its derived classes and should not provide the service to other clients.
* avoid including protected data members in base class. (include ~~[illegible]~~ non-private member functions that access private data members)
* if derived-class constructor calls one of its base-class constructors with arguments, the number and types of parameters specified in one of the base-class constructor definitions must match exactly.
* including a base-class member function with a different signature in derived class hides the base-class version of the function.

⊛ when an object of a derived class is created, first the constructors for the base class's member objects execute, then the base-class construct
executes, then the constructors for the derived class's member objects execute, then the derived class's constructor executes.

⊛ Destructors are called in the reverse of the order in which their corresponding constructors are called.

⊛ "uses a" relationship = (e.g., a function uses an object - - - . inheritance type)

⊛ "knows a" relationship = (association) e.g., one object is said to have a knows a relationship with the other object.

Base-class member access specifier	(type of inheritance) in derived class		
	public inheritance	protected inheritance	private inheritance
public	public (by non-static, friend & non-member functions)	protected (by non-static, friend functions)	private (by non-static, friend) functions
protected	protected (by non-static, friend, - -)	protected (non-static, friend - -)	private (non-static, friend - -)
private	Hidden in derived class (by non-static, friend function through public, protected functions of base class)	Hidden in - - $\Longleftrightarrow$ - - -	Hidden in - - - $\Longleftrightarrow$ - - -

* polymorphism enables us to "program ~~in~~ in general"
* an object of a derived class can be treated as an object of its base class.

(a program can create an array of base-class pointers that point to objects of many derived-class types)

(we can assign address of derived-class object to a base-class pointer)

* Assigning the address of a base-class object to a derived class pointer is error
[without an explicit cast, i.e., (downcasting) casting base-class pointers to derived-class pointers].

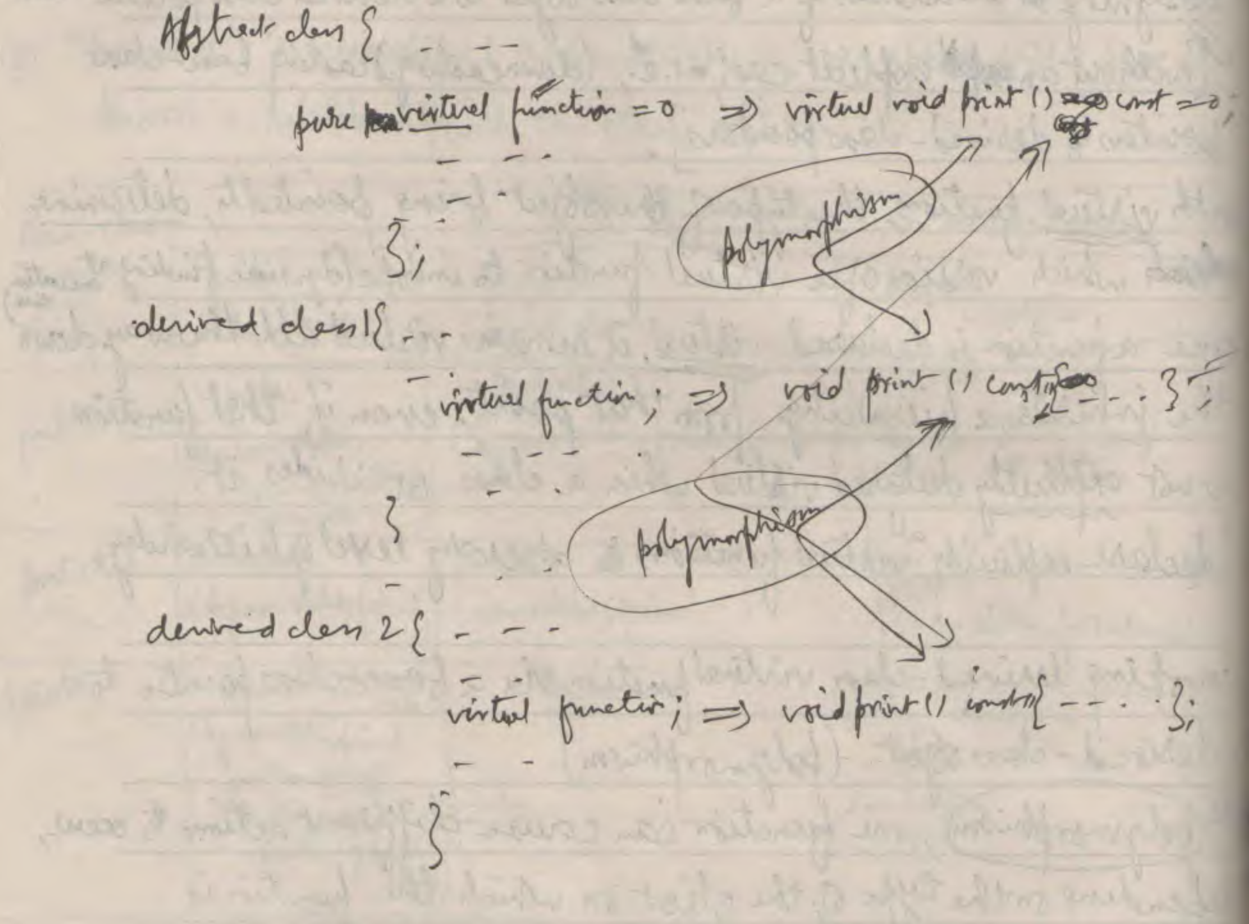
* with virtual functions, the type of the object being pointed to, determines ~~which~~ which version of a virtual function to invoke (dynamic binding at execution time)
* once a function is declared virtual, it remains virtual all the way down the inheritance hierarchy from that point, even if that function is not explicitly declared virtual when a class overrides it.
* declare explicitly all virtual functions at every level of hierarchy.

* invoking derived-class virtual function via a base-class pointer to a derived-class object (polymorphism)
* with polymorphism, one function can cause different actions to occur, depending on the type of the object on which the function is invoked.

⊛ ~~Abstract base~~
⊛ Abstract classes don't instantiate any objects and must have at least one pure virtual function (i.e., =0 initializer).

⊛ - Concrete classes can instantiate any objects.

* a virtual function has an implementation whereas, a pure virtual function does not ~~provide any~~ have implementation.



```
Abstract class { - - -
        pure virtual function = 0   =>  virtual void print() const = 0;
          - - -
          - -
   };
derived class1 { - -
        virtual function;  =>  void print() const { - - - };
          - - -
          - -
   }
derived class 2 { - - -
        virtual function;  =>  void print() const { - - - };
          - -
   }
```

* when C++ compiles a class that has one or more virtual functions, it [illegible] builds a virtual function table (vtable) for that class. An executing program uses the vtable to select the proper function implementation each time a virtual function of that class is called.

(*) if a class has virtual functions, provide a virtual destructor, even if one is not required for the class.

(*) Constructors can (not) be virtual.

= FUNCTION TEMPLATE =

```
template < class T >        = CLASS TEMPLATE
  -> template < typename   ElementType >
     template < class BorderType, class FillType >
```

(overloaded functions normally are used to perform similar operations on different types of data. If operations are identical for each type, they can be performed more compactly using function templates)

```
template < class T , int elements >   // note nontype parameter
template < class T = type >
                  -> type parameter can be specify a default type.
```

* class template enables type-specific versions of generic classes to be instantiated.

[illegible] friend ⇒ $\boxed{\text{template <class T> class X}}$ and friend void f1();

-> class template for class X

makes f1 function a friend of every class-template

- . . .

* for clarity, avoid using exception handling for purposes other than error handling.

Common Gateway Interface (CGI) / CGI Script –

- a standard for enabling apps (CGI programs / CGI scripts) to interact with web servers and (indirectly) with clients (e.g. web browsers).
- to generate (dynamic) web content using client input, DB and other info. services. (programatically)
- for use with HTTPd web server, Apache HTTP Server ✓

To execute program, we place the compiled C++ executable file in webserver's cgi-bin directory. (you may have to change .exe to .cgi) and type:

http://localhost/cgi-bin/Localtime.cgi
$\hookrightarrow$ hostname or IP address $\longrightarrow$ filename

```
# include <iostream>
~~using std::cout;~~
#include <ctime>
int main() {
        time_t  currentTime;  //variable for storing time
        cout << "Content-Type: text/html \n\n";
        ~~cout << "<?xml version=\"1.0\" ?>"~~
           << "<!DOCTYPE html Public \"-//W3C//DTD XHTML 1.0"
           << "Transitional//EN\" \"http://www.w3.org/TR/xhtml1"
           << "/DTD/xhtml1-transitional.dtd\">";
      time (&currentTime);
      cout << "html xmlns = \"http://www.w3.org/1999/xhtml\">"
           << "<head><title> current time date and </title></head>"
           << "<body><p>"
    →      << asctime ( localtime (&currentTime))
           << "</p></body></html>";
      return 0;
  }
```

XHTML form elements:-

name
input → text / password / checkbox / radio / button / submit / image / reset / file / hidden

select

textarea

preprocessor ⇒ $\boxed{\text{\#define identifier replacement-text}}$ → creates symbolic constants

e.g. #define PI 3.14159

#define CIRCLE_AREA(x) (PI * (x) * (x))

Conditional compilation ⇒

```
#ifndef ..
    #define ..
#endif
```

$\boxed{\text{\#error tokens}}$ e.g., #error 1 - Out of range error

$\boxed{\text{\#pragma tokens}}$

$\boxed{\text{\#define id token1\#\#token2}}$

#define TOKENCONCAT(x,y) x ##y

→ ## operator concatenates two tokens

line number ⇒ #line 100 "file1.cpp"

Predefined Symbolic Constants =

Assertions ⇒ ~~defined~~ assert macro - defined in <cassert> header file - tests value of an expression.

e.g., assert(x <= 10);

Data structures DS

- Self-referential classes (contains pointer member that points to a class object of the same class type)

```
class Node {
public:
        Node(int);
        void setData(int);
        int getData() const;
        void setNextPtr(Node *);
        Node *getNextPtr() const;
private:
        int data;
        Node *nextPtr;
};
```

- Dynamic memory allocation = ~~aa~~ (to create and maintain dynamic DS)
  new, delete

```
                listnode.h
#ifndef LISTNODE_H
# define LISTNODE_H
template < class NODETYPE > class List; //class template for class List
template < class NODETYPE >
   class ListNode {   friend class List <NODETYPE > ;      //make list a friend
   public:
              ListNode ( const NODETYPE & );
              NODETYPE  getData ( ) const ;
         private:  NODETYPE   date ;                    → next node in list
              ✓ ListNode < NODETYPE > * nextPtr ;  } ;
──────────────────────────────────────────────
   template < class NODETYPE >
  ListNode < NODETYPE > : : ListNode ( Const NODETYPE & info ) : date (info), nextPtr (
   {      }
──────────────────────────────
   template < class NODETYPE >
   NODETYPE  ListNode< NODETYPE > : : getData ( ) const
   { return date ;
   }
 # endif
```

Standard Template Library (STL)

(Reusable components that implement many common Datastructures and algorithms)

* three component -

1. containers (popular templatized datastructures)

   types: = (first class containers, adapters, near containers)

~~iterators~~ example:- vector (dynamically resizable array)
list (a linked list)
deque (a double-ended queue)

2. iterator = (have properties similar to those of pointers, are used by programs to manipulate the STL-container elements)

3. Algorithm = (functions that perform such common data manipulations as searching, sorting, and comparing elements (or entire data structures).)

1. Containers := categories =

category — (i). sequence/sequential (linear data structures like vector, link list, deque)
rst-class — (ii) Associative (non-linear container that typically can locate elements stored in the containers quickly. Such containers can store sets of values or key/value pairs.)

(iii) container adapter (enables a program to view a sequential container in a constrained manner)

near containers: (C-like arrays, string, bitset, valarray) standard library container classes =

vector (rapid insertions and deletions at back,
direct access to any element.)

deque (rapid insertions and deletions at front or back.
direct access to any element.)

list (doubly linked list, rapid insertion and deletion anywhere)

---

set (rapid lookup, no duplicates allowed)
multiset (rapid lookup, duplicate allowed)
map (one-to-one mapping, no duplicates allowed, rapid key-based lookup)
multimap (one-to-many mapping, duplicates allowed, rapid key-based lookup)

---

stack (last-in-first-out (LIFO))
queue (first-in-first-out (FIFO))
priority_queue (highest priority element is always the first element out)

---

header files:-
<vector>, <list>, <deque>, <queue>, <stack>, <map>, <set>, <bitset

Common member functions for all STL containers =

(default constructor, Copy constructor, destructor, empty, max_size,
size, operator=, operator<, operator<=, operator>, operator>=,
operator==, operator!=, swap).

* only in first-class containers =
~~[illegible]~~
(begin, end, rbegin, rend, erase, clear)

* typedef (to create synonyms/aliases for lengthy type names) found in first-class containers
in containers:
— used in generic declaration of variables, parameters to functions and return values from functions.

{value_type, reference, const_reference, pointer, iterator, const_iterator,
reverse_iterator, const_reverse_iterator, difference_type, size_type}

iterator category hierarchy =

$$\begin{tikzcd}
\text{input} \arrow[dr, dash] & & \text{output} \arrow[dl, dash] \\
& \text{forward} \arrow[d, dash] & \\
& \text{bidirectional} \arrow[d, dash] & \\
& \text{random access} &
\end{tikzcd}$$

```
#include <iostream>
  using namespace std;
#include <iterator> ⟹ →
int main ()
{  cout<< "enter two integers:";
      istream_iterator <int>  inputInt (cin) ; ⟶
    int number1 = *inputInt ;
        ++inputInt ;
    int number2 = *inputInt ;
   ostream_iterator <int> outputInt(cout); ⟶
      cout<< "total:.." ;
     *outputInt = number1 +number2 ;
          cout << endl ;
       return 0 ;
  }
```