C++ Data Structures
Prerequisites

CHAPTER 6: ABSTRACT DATA TYPES

(STRING/ARRAY/VECTOR/ITERATOR/LISTS)

DR. ERIC CHOU IEEE SENIOR MEMBER

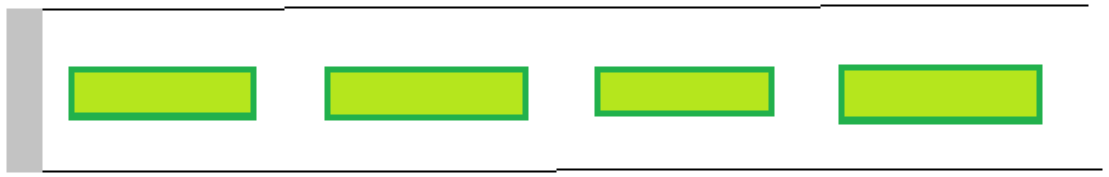
Objectives

- What is abstract Data Type?
- Conceptual Model and Physical Implementation.
- Relationship of ADT and Data Structures.
- String/Vector/Stack/Queue/DeQue

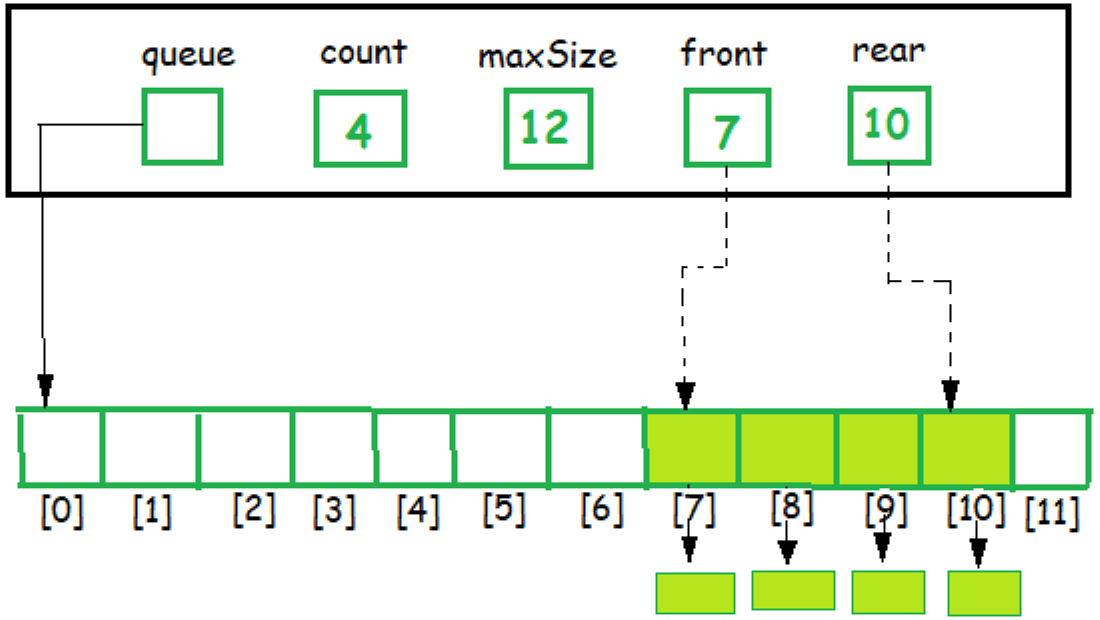
LECTURE 1

Abstract Data Type

ADT = Type + Function names + Behaviour of each Function



a) Conceptual



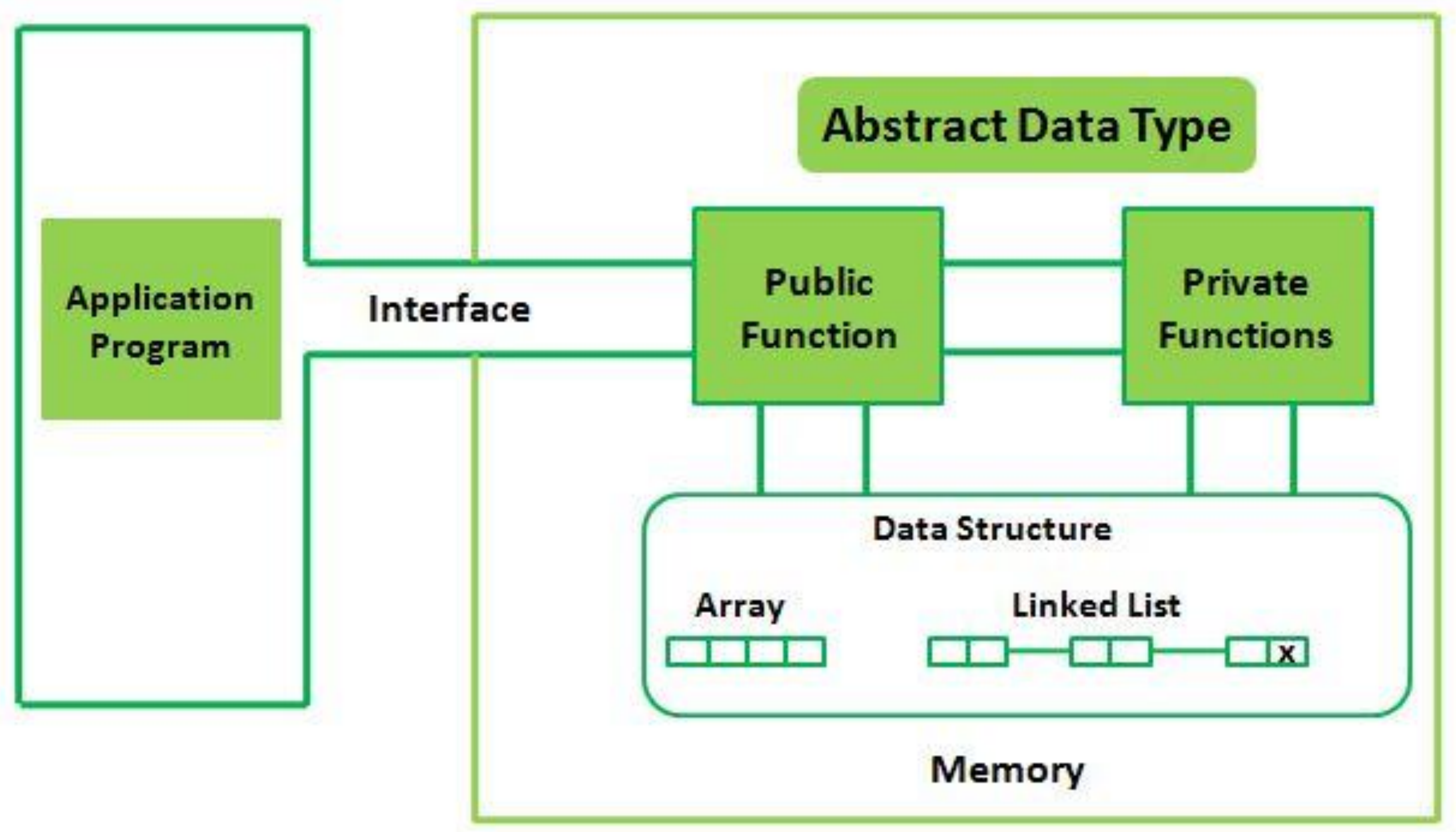
b) Physical Structures

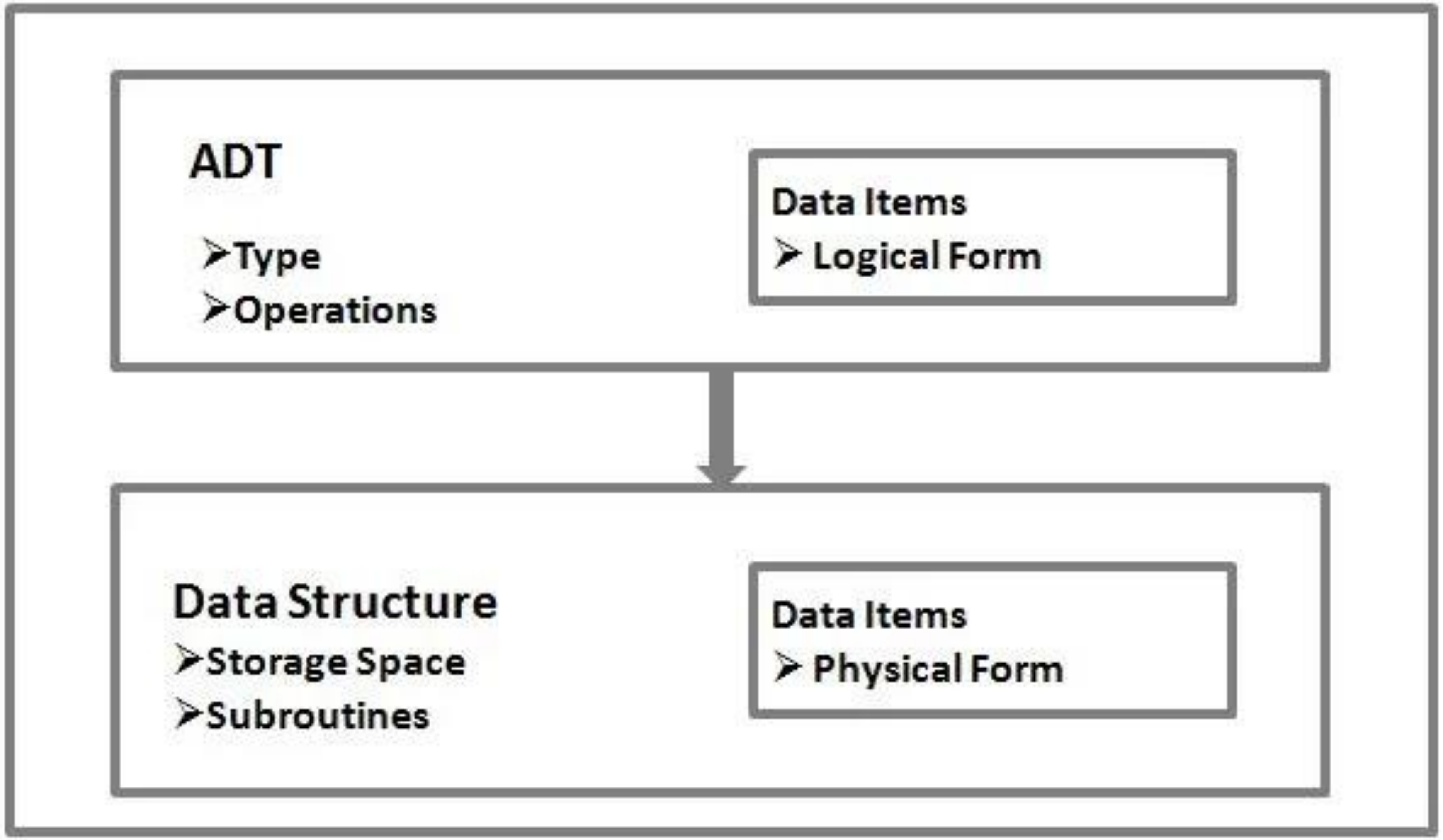
Abstract Data Type

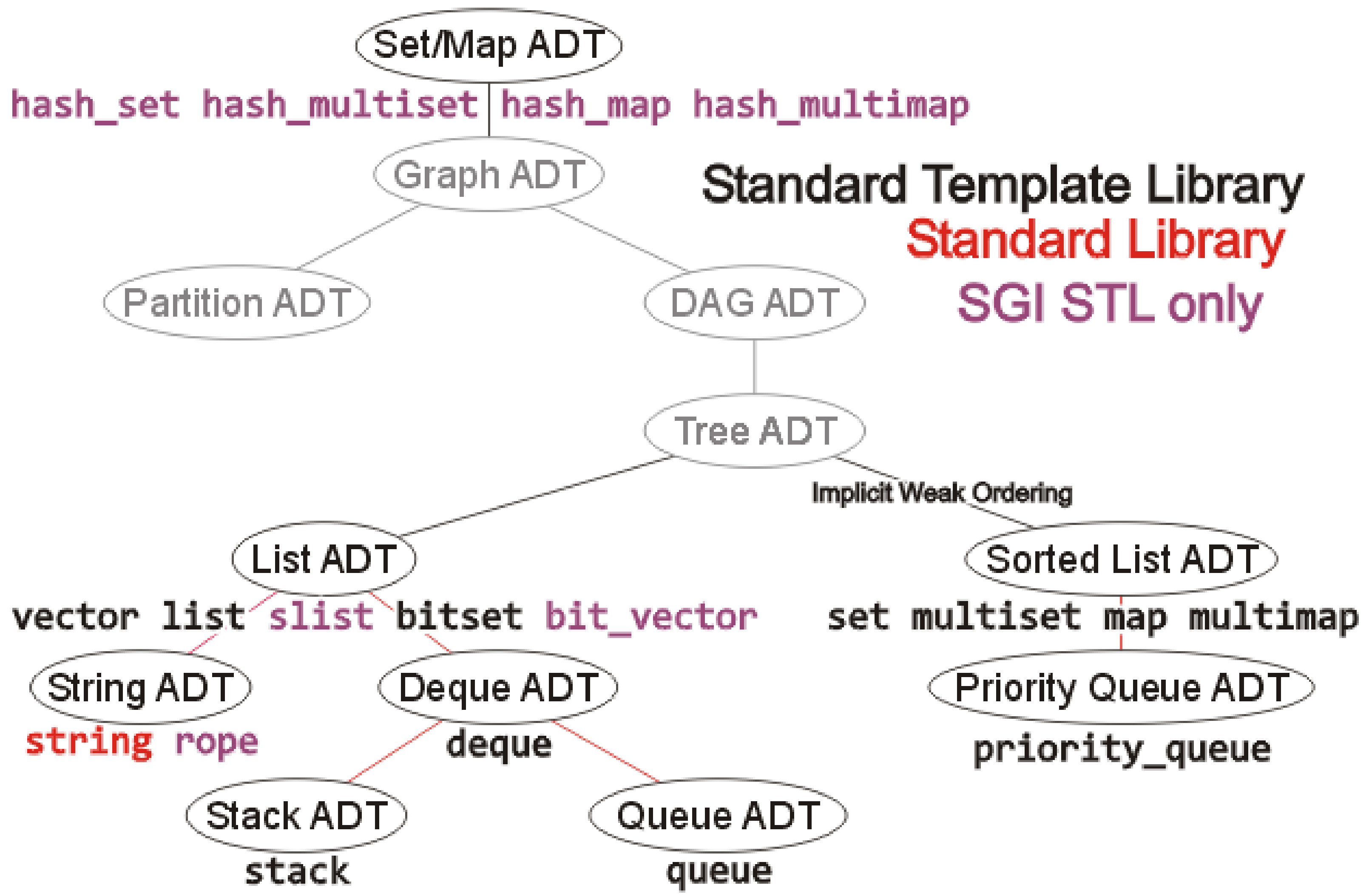
- In computer science, abstract Data types (**ADT**) is a class (or type) for objects whose behavior of each function is defined by a set of values and a set of operations.
- In another way, you can say that abstract data types (ADT) are a mathematical model for data types where the data types defined by its semantics(behavior) from the pint of view of a user of the data.
- therefore, in case of possible values, possible operations on data oof these types and the behavior of the operations.

Abstract Data Type

- ADT only mentions which operation is to be performed but not this operation how will be implemented.
- How data will be organized in the memory and what algorithms will be used for operation it does not specify in ADT.
- It is called abstract because it gives an independent-implementation view.
- Therefore, only the essentials and hiding details providing process are known as an abstraction.

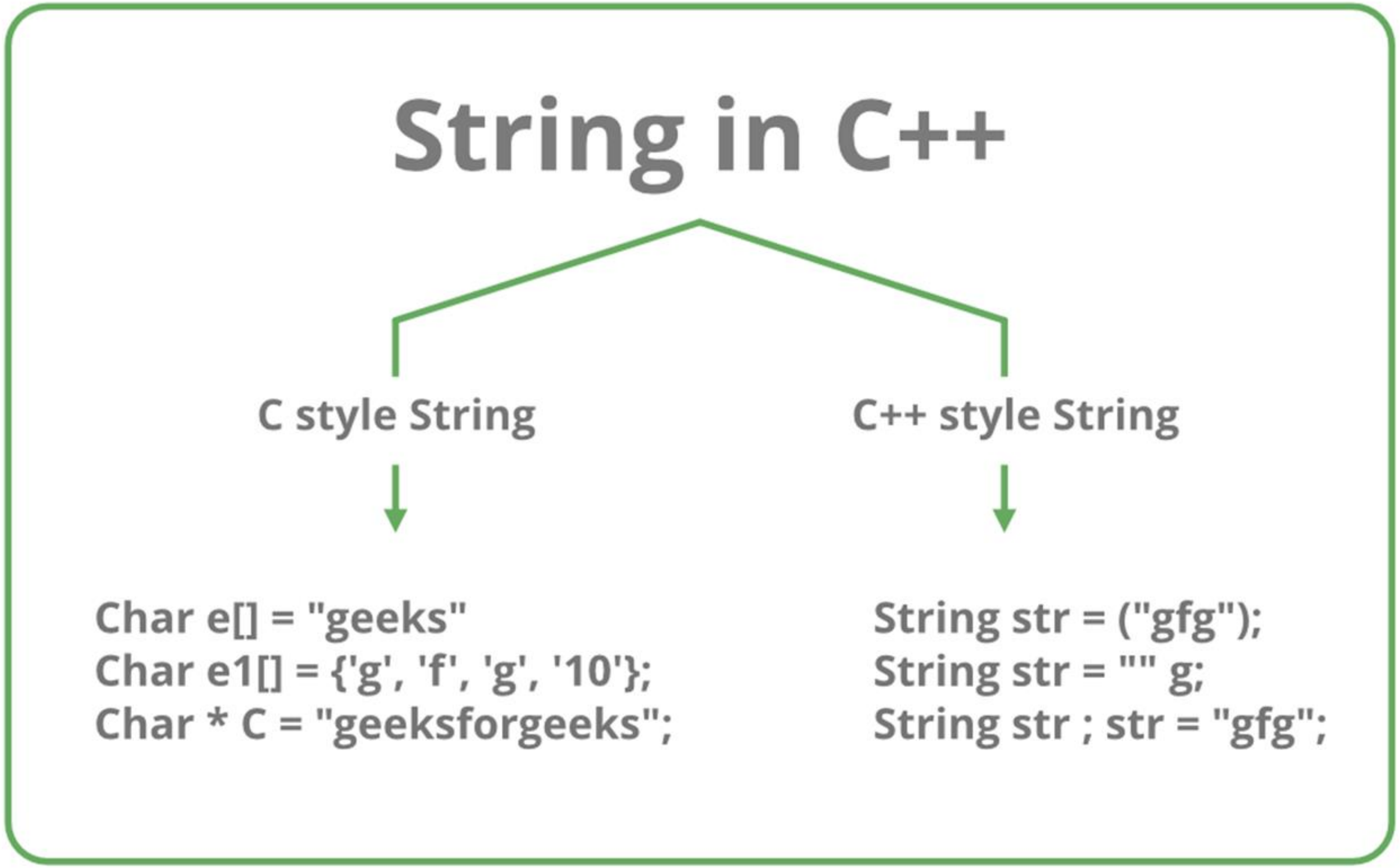

Application Program
Interface
Abstract Data Type
Public Function
Private Functions
Data Structure
Array
Linked List
x
Memory


ADT
➢Type
➢Operations
Data Items
➢ Logical Form
Data Structure
➢Storage Space
➢Subroutines
Data Items
➢ Physical Form


Set/Map ADT
hash_set hash_multiset hash_map hash_multimap
Graph ADT
Standard Template Library
Standard Library
SGI STL only
Partition ADT
DAG ADT
Tree ADT
Implicit Weak Ordering
List ADT
Sorted List ADT
vector list slist bitset bit_vector
set multiset map multimap
String ADT
string rope
Deque ADT
deque
Priority Queue ADT
priority_queue
Stack ADT
stack
Queue ADT
queue

LECTURE 1

String



```
Char e[] = "geeks"
Char e1[] = {'g', 'f', 'g', '10'};
Char * C = "geeksforgeeks";
```

```
String str = ("gfg");
String str = "" g;
String str ; str = "gfg";
```

The C++ `string` Class

- Special datatype supports working with strings

```
#include <string>
```

- Can define `string` variables in programs:

```
string firstName, lastName;
```

- Can receive values with assignment operator:

```
firstName = "George";
lastName = "Washington";
```

- Can be displayed via `cout`

```
cout << firstName << " " << lastName;
```

Input into a `string` Object

- Use `getline` function to put a line of input, possibly including spaces, into a string:

```
string address;
cout << "Enter your address: ";
getline(cin,address);
```

`string` Comparison

- Can use relational operators directly to compare string objects:

```
string str1 = "George",
            str2 = "Georgia";
if (str1 < str2)
     cout << str1 << " is less than "
            << str2;
```

- Comparison is performed similar to `strcmp` function. Result is `true` or `false`

Constructors

- **string ()**
  - creates an empty string ("")
- **string (other_string)**
  - creates a string identical to other_string
- **string (other_string, position, count)**
  - creates a string that contains count characters from other_string, starting at position. If count is missing (only the first two arguments are given), all the characters from other_string, starting at position and going to the end of other_string, are included in the new string.
- **string (count, character)**
  - create a string containing character repeated count times

Other Definitions of C++ `strings`

Definition	**Meaning**
`string name;`	defines an empty string object
`string myname("Chris");`	defines a string and initializes it
`string yourname(myname);`	defines a string and initializes it
`string aname(myname, 3);`	defines a string and initializes it with first 3 characters of myname
`string verb(myname,3,2);`	defines a string and initializes it with 2 characters from myname starting at position 3
`string noname('A', 5);`	defines string and initializes it to 5 'A's

Examples:

```
string s1;                  //    s1 = ""
string s2("abcdef");        //    s2 = "abcdef"
string s3(s2);              //    s3 = "abcdef"
string s4(s2, 1);           //    s4 = "bcdef"
string s5(s2, 3, 2);        //    s5 = "de"
string s6(10, '-');         //    s6 = "----------"
```

string Operators

OPERATOR	MEANING
`>>`	extracts characters from stream up to whitespace, insert into string
`<<`	inserts string into stream
`=`	assigns string on right to string object on left
`+=`	appends string on right to end of contents on left
`+`	concatenates two strings
`[]`	references character in string using array notation
`>, >=, <, <=, ==, !=`	relational operators for string comparison. Return true or false

Operators Defined for `string`

Assign =

```
string s1;
string s2;

s1 = s2; // the contents of s2 is copied to s1
```

Append +=

```
string s1( "abc" );
string s2( "def" );

s1 += s2; // s1 = "abcdef" now
```

Operators Defined for `string`

Indexing []

```
string s( "def" );
char c = s[2];      // c = 'f' now
s[0] = s[1];        // s = "eef" now
```

Concatenate +

```
string s1("abc");
string s2("def");
string s3;

s3 = s1 + s2; // s3 = "abcdef" now
```

Operators Defined for `string`

Equality ==

```
string s1("abc");
string s2("def");
string s3("abc");

bool flag1 = (s1 == s2); // flag1 = false now
bool flag2 = (s1 == s3); // flag2 = true now
```

Inequality !=

- the inverse of equality

Operators Defined for `string`

Comparison <, >, <=, >=

- performs case-insensitive comparison

```
string s1 = "abc";
string s2 = "ABC";
string s3 = "abcdef";

bool flag1 = (s1 < s2); // flag1 = false now
bool flag2 = (s2 < s3); // flag2 = true now
```

string Operators

```
string word1, phrase;
string word2 = " Dog";
cin >> word1;
phrase = word1 + word2;
phrase += " on a bun";
for (int i = 0; i < 16; i++)
      cout << phrase[i]; // displays
```

`string` Member Functions

- Are behind many overloaded operators
- Categories:
  - assignment: `assign, copy, data`
  - modification: `append, clear, erase, insert, replace, swap`
  - space management: `capacity, empty, length, resize, size`
  - substrings: `find, substr`
  - comparison: `compare`

Constant Member Functions

`const char *data()`

- returns a C-style null-terminated string of characters representing the contents of the string

`unsigned int length()`

- returns the length of the string

`unsigned int size()`

- returns the length of the string (i.e., same as the length function)

`bool empty()`

- returns true if the string is empty, false otherwise

string Member Functions

```
string word1, word2, phrase;
cin >> word1;
word2.assign(" Dog");
phrase.append(word1);
phrase.append(word2);
phrase.append(" with mustard relish", 13);
phrase.insert(8, "on a bun ");
cout << phrase << endl;
```

Member Functions

void swap (other_string)

- swaps the contents of this string with the contents of other_string.

```
  string s1( "abc" );
  string s2( "def" );
  s1.swap( s2 ); // s1 = "def", s2 = "abc" now
```

string & append (other_string)

- appends other_string to this string, and returns a reference to the result string.

string & insert (position, other_string)

- inserts other_string into this string at the given position, and returns a reference to the result string.

Member Functions

string & erase (position, count)

- removes count characters from this string, starting with the character at the given position. If count is ommitted (only one argument is given), the characters up to the end of the string are removed. If both position and count are omitted (no arguments are given), the string is cleared (it becomes the empty string). A reference to the result string is returned.

unsigned int find (other_string, position)

- finds other_string inside this string and returns its position. If position is given, the search starts there in this string, otherwise it starts at the beginning of this string.

string substr (position, count)

- returns the substring starting at position and of length count from this string

string2.cpp

```cpp
/* To demonstrate std::string */
#include<iostream>
#include<string>

using namespace std;
int main()
{
    /* s becomes object of class string. */
    string s;

    /* Initializing with a value. */
    s = "HELLO";

    /* Printing the value */
    cout << s;

    return 0;
}
```

```
HELLO
```

LECTURE 1

C-String

Character Testing

require cctype header file

FUNCTION	MEANING
`isalpha`	`true` if arg. is a letter, `false` otherwise
`isalnum`	`true` if arg. is a letter or digit, `false` otherwise
`isdigit`	`true` if arg. is a digit 0-9, `false` otherwise
`islower`	`true` if arg. is lowercase letter, `false` otherwise
`isprint`	`true` if arg. is a printable character, `false` otherwise
`ispunct`	`true` if arg. is a punctuation character, `false` otherwise
`isupper`	`true` if arg. is an uppercase letter, `false` otherwise
`isspace`	`true` if arg. is a whitespace character, `false` otherwise

Similar to

```
int  isdigit(char  ch)
 { if  (48<=ch && ch<=57)
      return  4;
   else  return  0;
 }

int  isupper(char  ch)
 { if  (65<=ch && ch<=90)
        return  1;
   else  return  0;
 }

int  islower(char  ch)
 { if  (97<=ch && ch<=122)
         return  2;
         else  return  0;
 }
```

```
int  isspace(char  ch)
 { if  (8<=ch && ch<=13)
              return  8;
  else  return  0;
 }

int  isalpha(char  ch)
 { if (isupper(ch))
       return  1;
  else  if (islower(ch))
       return  2;
  else  return  0;
 }
```

Similar to

```
int  isalnum(char  ch)
       { if (isupper(ch))
            return  1;
       else  if (islower(ch))
            return  2;
       else  if (isdigit(ch))
           return  4;
       else  return  0;
       }

int  ispunct(char  ch)
       { if  (33<=ch && ch<=126 && ! isdigit(ch) &&
                         ! isupper(ch) && ! islower(ch))
                  return  16;
         else  return  0;
       }
```

```
int  isprint(char  ch)
       { if (isupper(ch))
                      return  1;
       else  if (islower(ch))
              return  2;
       else  if (isdigit(ch))
              return  4;
       else  if (ispunct(ch))
             return  16;
       else  if  (ch==32)
             return  64;
       else  return  0;
       }
```

Character Case Conversion

require `cctype` header file

functions:

`toupper`: if `char` argument is lowercase letter, return uppercase equivalent; otherwise, return input unchanged

```
char greeting[] = "Hello!";
cout << toupper[0];
cout << toupper[1];
cout << toupper[5];
```

Character Case Conversion

functions:

`tolower`: if `char` argument is uppercase letter, return lowercase equivalent; otherwise, return input unchanged

```
char greeting[] = "Hello!";
cout << tolower[0];
cout << tolower[1];
cout << tolower[5];
```

Review of the Internal Storage of C-Strings

<u>C-string</u>: sequence of characters stored in adjacent memory locations and terminated by `NULL` character

<u>String literal</u> (<u>string constant</u>): sequence of characters enclosed in double quotes " " : `"Hi there!"`

H	i		t	h	e	r	e	!	\0

Review of the Internal Storage of C-Strings

- Array of `chars` can be used to define storage for string:

```
const int SIZE = 20;
char city[SIZE];
```

- Leave room for `NULL` at end
- Can enter a value using `cin` or `>>`
  - Input is whitespace-terminated
  - No check to see if enough space
- For input containing whitespace, and to control amount of input, use `cin.getline()`

Library Functions for Working with C-Strings

- require `cstring` header file
- functions take one or more C-strings as arguments. Can use:
  - C-string name
  - pointer to C-string
  - literal string

Library Functions for Working with C-Strings

Functions:

– **`strlen(str)`:** returns length of C-string `str`

```
char city[SIZE] = "Missoula";
cout << strlen(city); // prints 8
```

– **`strcat(str1, str2)`:** appends `str2` to the end of `str1`

```
char location[SIZE] = "Missoula, ";
char state[3] = "MT";
strcat(location, state);
// location now has "Missoula, MT"
```

Library Functions for Working with C-Strings

Functions:

– **`strcpy(str1, str2)`:** copies `str2` to `str1`

```
const int SIZE = 20;
char fname[SIZE] = "Maureen", name[SIZE];
strcpy(name, fname);
```

Note: `strcat` and `strcpy` perform no bounds checking to determine if there is enough space in receiving character array to hold the string it is being assigned.

C-string Inside a C-string

- **Function:**
  - `strstr(str1, str2)`: finds the first occurrence of `str2` in `str1`. Returns an address to match, or `NULL` if no match.

```
char river[] = "Wabash";
char word[] = "aba";
cout << strstr(state, word);
// displays "abash"
```

String/Numeric Conversion Functions

require `cstdlib` header file

FUNCTION	PARAMETER	ACTION
`atoi`	C-string	converts C-string to an `int` value, returns the value
`atol`	C-string	converts C-string to a `long` value, returns the value
`atof`	C-string	converts C-string to a `double` value, returns the value
`itoa`	`int`, C-string, `int`	converts 1st `int` parameter to a C-string, stores it in 2nd parameter. 3rd parameter is base of converted value

String/Numeric Conversion Functions

```
int iNum;
long lNum;
double dNum;
char intChar[10];
iNum = atoi("1234"); // puts 1234 in iNum
lNum = atol("5678"); // puts 5678 in lNum
dNum = atof("35.7"); // puts 35.7 in dNum
itoa(iNum, intChar, 8); // puts the string
     // "2322" (base 8 for 1234₁₀) in intChar
```

String/Numeric Conversion Functions - Notes

- if C-string contains non-digits, results are undefined
  - function may return result up to non-digit
  - function may return 0
- `itoa` does no bounds checking – make sure there is enough space to store the result

Writing Your Own C-String Handling Functions

- Designing C-String Handling Functions
  - Can perform bounds checking to ensure enough space for results
  - Can anticipate unexpected user input

```cpp
/* To demonstrate C style strings */
#include<iostream>
using namespace std;
int main()
{
    /* Null character has to be added explicitly */
    char str1[8] = {'H' , 'E' , 'L' , 'L' , 'O' ,
                    '-','1','\0' };

    /* Compiler implicitly adds Null character */
    char str2[] = "HELLO-2" ;

    /* Compiler implicitly adds Null character.
    Note that string literals are typically stored
    as read only */
    const char *str3 = "HELLO-3" ;

    cout << str1 << endl << str2 << endl << str3;
    return 0;
}
```

string1.cpp

```
HELLO-1
HELLO-2
HELLO-3
```

LECTURE 1

C-String – C++ String Conversion

Converting C-String to a std::string.

But why do we need this transformation? From a C string to a std::string? It is because

- Std::string manages its own space. So programmer don’t need to worry about memory , unlike C strings (Since they are array of characters)
- They are easy to operate. ‘+’ operator for concatenation, ‘=’ for assignment, can be compared using regular operators.
- **string::find()** and many other functions can be implemented on std::string and not on C-Strings so this becomes handy.
- Iterators can be used in std::string and not in C-strings.

```cpp
/* To demonstrate C style string to std::string */
#include<bits/stdc++.h>

using namespace std;
int main()
{
     /*Initializing a C-String */
     const char *a = "Testing";
     cout << "This is a C-String : "<< a << endl;

     /* This is how std::string s is assigned
     though a C string 'a' */
     string s(a);

     /* Now s is a std::string and a is a C-String */
     cout << "This is a std::string : "<< s << endl;
     return 0;
}
```

string3.cpp

```
This is a C-String : Testing
This is a std::string : Testing
```

Converting a std::string to a C style string

Why do we need this transformation? From std::string to a C string?

- It is because there are several powerful functions in header that makes our work very much easier.
- **atoi()** , **itoa()** , and many more functions work with C strings only.

You can think of other reasons too!

```cpp
/* To demonstrate std::string to C style string */
#include<iostream>
#include<string> /* This header contains string class */
using namespace std;
int main(){
    /* std::string initialized */
    string s = "Testing";
    cout << "This is a std::string : "<< s << endl;

    /* Address of first character of std::string is
    stored to char pointer a */
    char *a = &(s[0]);

    /* Now 'a' has address of starting character
    of string */
    printf("%s\n", a);
    return 0;
}
```

string4.cpp

```
This is a std::string : Testing
Testing
```

Converting a std::string to a C style string

- std::string also has a function c_str() that can be used to get a null terminated character array.
- Both C strings and std::strings have their own advantages. One should know conversion between them, to solve problems easily and effectively.

string5.cpp

```
/* To demonstrate std::string to C style string usingc_str() */
#include<bits/stdc++.h>
using namespace std;

int main(){
    /* std::string initialized */
    string s = "Testing";
    cout << "This is a std::string : "<< s << endl;

    // c_str returns null terminated array of characters
    const char *a = s.c_str();

    /* Now 'a' has address of starting character of string */
    printf("%s\n", a);
    return 0;
}
```

```
This is a std::string : Testing
Testing
```

Convert Data to String

- Convert primitive data types to **`string`**: **`std::to_string()`**
- Convert **`boolean`** data to **`string`**: user defined method

```
string &to_string(bool b){ return b ? “true” :
                                    “false”; }
```

- Convert an object to string:

  override `operator std::string()`

- Insert an object to the cout (any sstream):

  `ostream& operator<<(ostream& out, const Data& d);`

  // std namespace

std::to_string

```
string to_string (int val);
string to_string (long val);
string to_string (long long val);
string to_string (unsigned val);
string to_string (unsigned long val);
string to_string (unsigned long long val);
string to_string (float val);
string to_string (double val);
string to_string (long double val);
```

C/C++

C sprintf

Declaration

Following is the declaration for sprintf() function.

```
int sprintf(char *str, const char *format, ...)
```

Parameters

- **str** − This is the pointer to an array of char elements where the resulting C string is stored.
- **format** − This is the String that contains the text to be written to buffer. It can optionally contain embedded format tags that are replaced by the values specified in subsequent additional arguments and formatted as requested.

Format tag

prototype: **%[flags][width][.precision][length]specifier**, as explained below

string6.cpp

```
#include <cstdio>
#define _USE_MATH_DEFINES  // must be before #include <cmath>
#include <cmath>

using namespace std;

int main () {
   char str[80];

   sprintf(str, "Value of Pi = %f", M_PI);
   puts(str);
   return(0);
}
```

```
Value of Pi = 3.141593
```

Mathematical Expression	C++ Symbol	Decimal Representation
pi	M_PI	3.14159265358979323846
pi/2	M_PI_2	1.57079632679489661923
pi/4	M_PI_4	0.785398163397448309616
1/pi	M_1_PI	0.318309886183790671538
2/pi	M_2_PI	0.636619772367581343076
2/sqrt(pi)	M_2_SQRTPI	1.12837916709551257390
sqrt(2)	M_SQRT2	1.41421356237309504880
1/sqrt(2)	M_SQRT1_2	0.707106781186547524401
e	M_E	2.71828182845904523536
log_2(e)	M_LOG2E	1.44269504088896340736
log_10(e)	M_LOG10E	0.434294481903251827651
log_e(2)	M_LN2	0.693147180559945309417
log_e(10)	M_LN10	2.30258509299404568402

Object to String

Override operator std::string()

```
operator std::string() const{
    return std::to_string(month) + " / "
         + std::to_string(day)   + " / "
         + std::to_string(year);
}
```

cout Insertion

Convert a **Date** object a string to the **out** stream

```
std::ostream& operator<<(std::ostream& out, const Date& d){
    return out << std::string(d);
}
```

```cpp
#include <iostream>
#include <string>
struct Date{
    int year, month, day;
    Date(int y, int m, int d) : year{y}, month{m}, day{d} {}
    operator std::string() const{
        return std::to_string(month) + " / "
             + std::to_string(day)   + " / "
             + std::to_string(year);
    }
};

std::ostream& operator<<(std::ostream& out, const Date& d){
    return out << std::string(d);
}

int main(){
    Date holiday(1997, 2, 13);
    // Not using operator<< overload
    std::cout << std::string(holiday) << '\n';
    // Using operator<< overload
    std::cout << holiday << '\n';
}
```

string7.cpp

```
2 / 13 / 1997
2 / 13 / 1997
```

Searching and Substrings

- The string class supports simple searching and substring retrieval using the functions find(), rfind(), and substr(). The find member function takes a string and a position and begins searching the string from the given position for the first occurence of the given string. It returns the position of the first occurence of the string, or a special value, string::npos, that indicates that it did not find the substring.
- This is what the find function prototype would look like. (Note that I've used ints here for clarity, but they would actually be of the type "size_type", which is unsigned.)

```
    int find(string pattern, int position);
```

string8.cpp

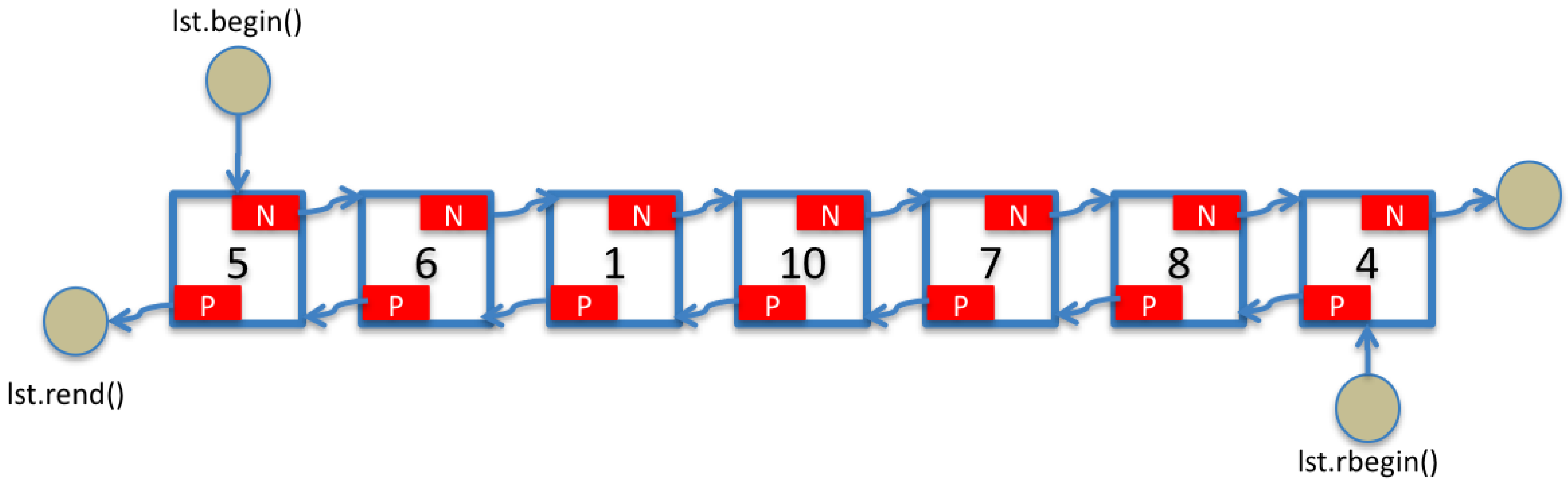
```
#include <iostream>
using namespace std;
int main(){
    string input;
    int i = 0;
    int cat_appearances = 0;

    getline(cin, input, '\n');

    for(i = input.find("cat", 0); i != string::npos; i = input.find("cat", i)){
        cat_appearances++;
        i++;  // Move past the last discovered instance to avoid finding same
            // string
    }
    cout<<cat_appearances;
}
```

Input:
cat catch a ball from catchup
Output:
3

Not found sumbol:
`string::npos`

LECTURE 1

List



List in C++ is linked list.

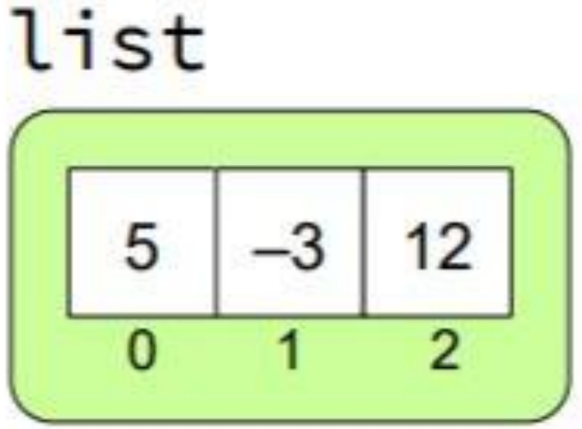
List

- Please refer to Chapter 5.

LECTURE 1

Vector

```
vector<int> list(3);
list[0] = 5;
list[1] = -3;
list[2] = 12;
```



Vector in C++ is dynamic array.

Why Use Vectors in C++?

- Vectors C++ are preferable when managing ever-changing data elements.
- It is handy if you don't know how big the data is beforehand since you don't need to set the maximum size of the container. Since it's possible to resize C++ vectors, it offers better flexibility to handle dynamic elements.
- C++ vectors offer excellent efficiency. It is a **`template`** class, which means no more typing in the same code to handle different data.

Why Use Vectors in C++?

- If you use vectors, you can copy and assign other vectors with ease. There are different ways to do that: using the iterative method, assignment operator =, an in-built function, or passing vector as a constructor.
- In C++ vectors, automatic reallocation happens whenever the total amount of memory is used. This reallocation relates to how size and capacity function works.

How to Create C++ Vectors?

Vectors in C++ work by declaring which program uses them. The common syntax look like this:

```
vector <type> variable(elements)
```

For example:

```
vector <int> rooms(9);
```

How to Create C++ Vectors?

Let's break it down:

• **`type`**	defines a data type stored in a vector (e.g., <int>, <double> or <string>)
• **`variable`**	is a name that you choose for the data
• **`elements`**	specified the number of elements for the data

How to Create C++ Vectors

- It is mandatory to determine the type and variable name. However, the number of elements is optional.
- Basically, all the data elements are stored in contiguous storage. Whenever you want to access or move through the data, you can use iterators.
- The data elements in C++ vectors are inserted at the end. Use modifiers to insert new elements or delete existing ones.

Iterators

An iterator allows you to access the data elements stored within the C++ vector. It is an object that functions as a pointer. There are five types of iterators in C++: input, output, forward, bidirectional, and random access.

C++ vectors support random access iterators. Here are a few function you may use with iterators for C++ vectors:

- `vector::begin()` returns an iterator to point at the first element of a C++ vector.
- `vector::end()` returns an iterator to point at past-the-end element of a C++ vector.
- `vector::cbegin()` is similar to `vector::begin()`, but without the ability to modify the content.
- `vector::cend()` issimilar to `vector::end()` but can't modify the content.

Modifiers

As its name suggests, you can use a modifier to change the meaning of a specified type of data. Here are some modifiers you can use in C++ vectors:

- `vector::push_back()` pushes elements from the back.
- `vector::insert()` inserts new elements to a specified location.
- `vector::pop_back()` removes elements from the back.
- `vector::erase()` removes a range of elements from a specified location.
- `vector::clear()` removes all elements.

Start with default value

vector1.cpp

```cpp
#include <vector>
using namespace std;
int main() {
    // Vector with 5 integers
    // Default value of integers will be 0.
    std::vector <int> vecOfInts(5);
    for (int x: vecOfInts)
        std::cout << x << std::endl;
}
```

```
0
0
0
0
0
```

eC Learning Channel

Start with an array

vector2.cpp

```
#include <iostream>
#include <string>
#include <iostream>
#include <string>
#include <vector>
using namespace std;
int main() {
    // Array of string objects
    string arr[] = { "first", "sec", "third", "fourth" };
    // Vector with a string array
    vector<string> vecOfStr(arr, arr+sizeof(arr)/sizeof(string));
    for (string str: vecOfStr) cout << str << endl;
}
```

```
first
sec
third
fourth
```

Start with a list

vector3.cpp

```
#include <bits/stdc++.h>
using namespace std;
int main() {
    // std::list of 5 string objects
    list<string> listOfStr;
    listOfStr.push_back("first");
    listOfStr.push_back("sec");
    listOfStr.push_back("third");
    listOfStr.push_back("fourth");
    // Vector with std::list
    vector <string> vecOfStr(listOfStr.begin(), listOfStr.end());
    for (string str: vecOfStr)
        cout << str << endl;
}
```

```
first
sec
third
fourth
```

Start by copying from another vector

```
#include <bits/stdc++.h>
using namespace std;
int main() {
    vector <int> v {1,2,3,4,5};
    int n = v.size();
    cout << "Size of the vector is :" << n << endl;
    cout << "max_size:" << v.max_size() << endl;
    cout << "v.operator[]:";
    for (int i=0; i<v.size();i++) cout << v.operator[](i) << " ";

    cout << endl;
}
```

vector4.cpp

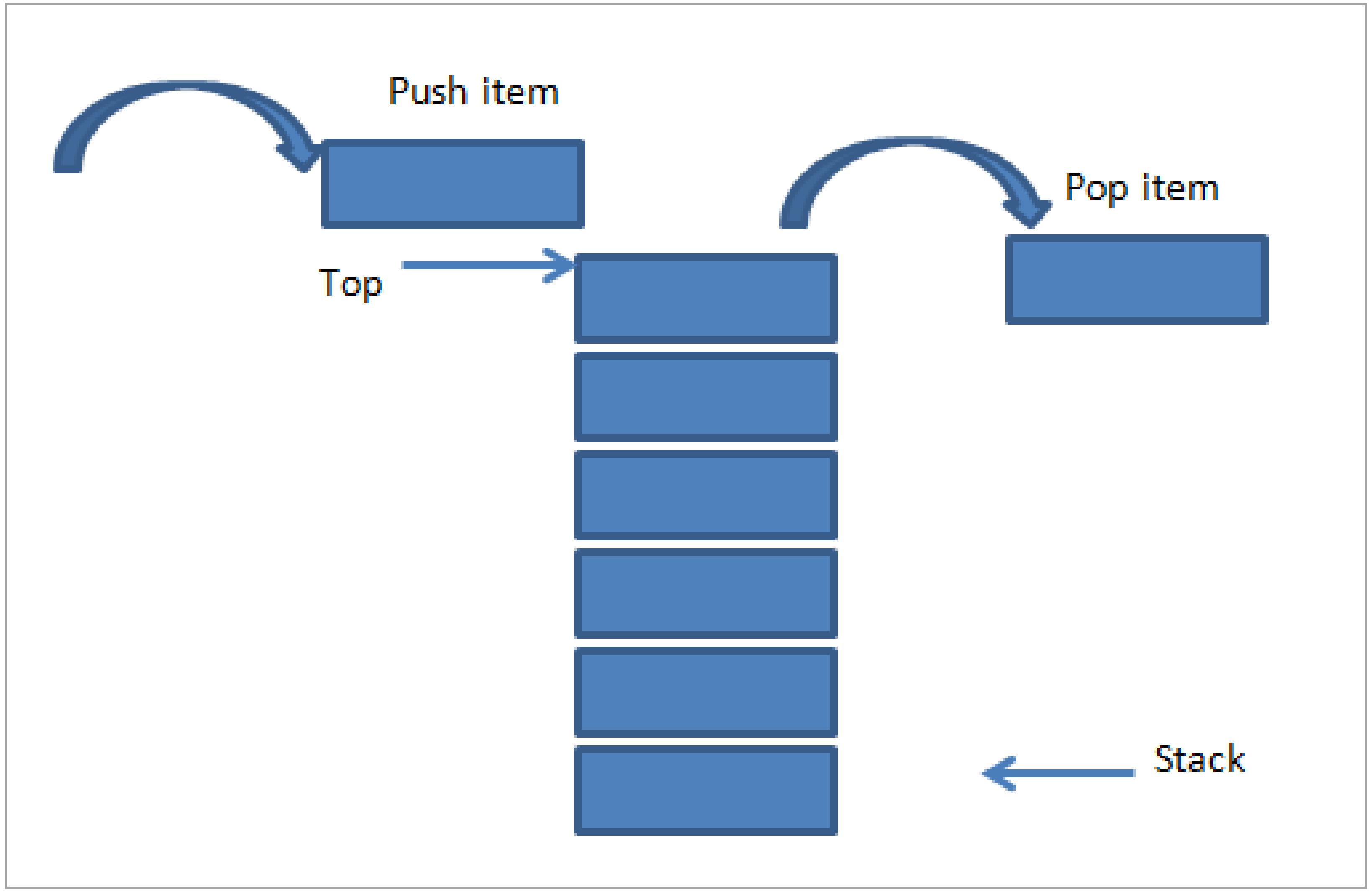
```
Size of the vector is :5
max_size:2305843009213693951
v.operator[]:1 2 3 4 5
```

C++ Vector: Useful Tips

- It is recommended to use C++ vector if your data elements are not predetermined.
- As a template class, C++ vectors offer better efficiency and reusability.
- Compared to arrays, there are more ways to copy vectors in C++.

LECTURE 1

Stack

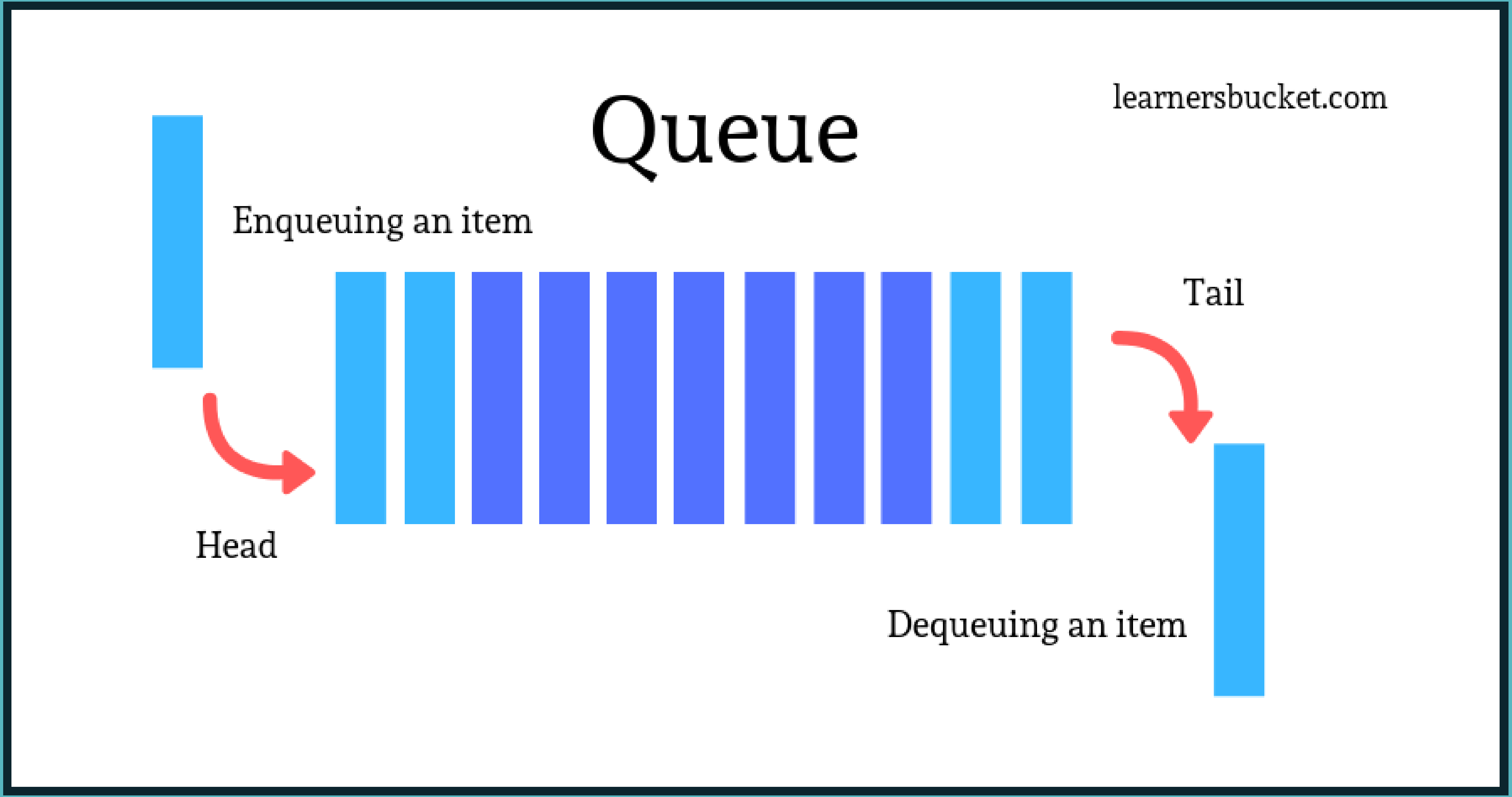

Push item
Pop item
Top
Stack

What is std::stack?

- A stack is a data structure that operates based on LIFO (Last In First Out) technique. The std::stack allows elements to be added and removed from one end only.
- The std::stack class is a container adapter. Container objects hold data of a similar data type. You can create a stack from various sequence containers. If no container is provided, the deque container will be used by default. Container adapters don't support iterators, so it can't be used to manipulate data.

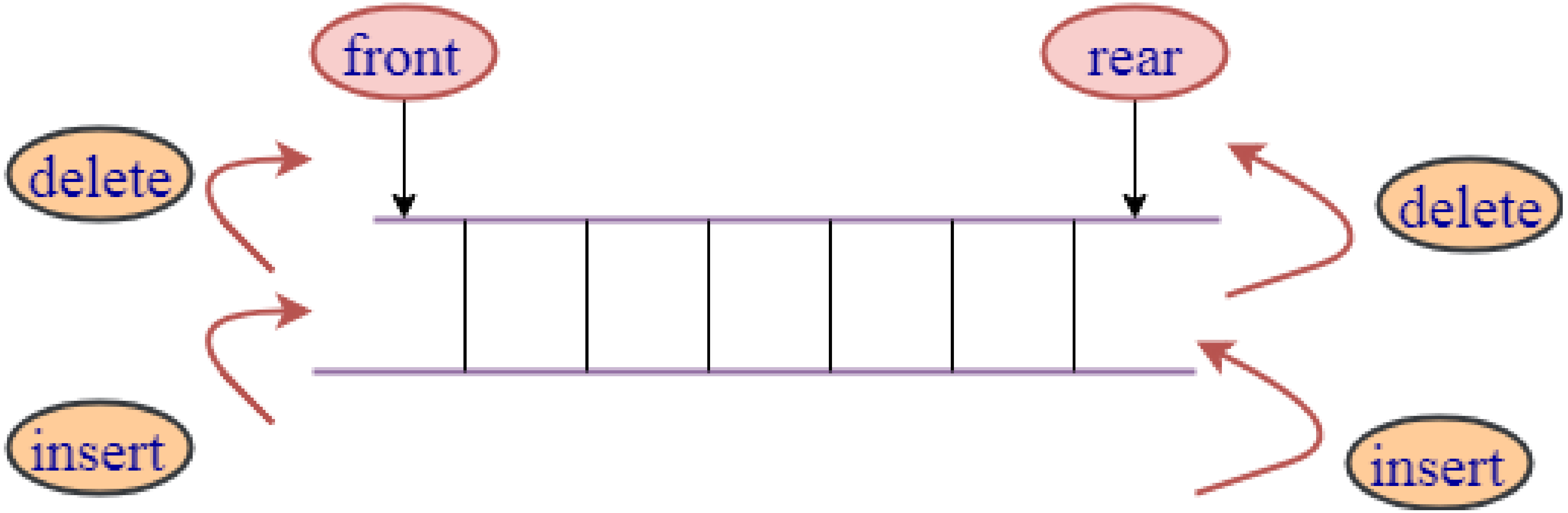
LECTURE 1

Queue


learnersbucket.com
Queue
Enqueuing an item
Tail
Head
Dequeuing an item

LECTURE 1

DeQueue


front
rear
delete
delete
insert
insert

DEQUE	STACK	QUEUE
size()	size()	size()
isEmpty()	isEmpty()	isEmpty()
Insert_First()	-	-
Insert_Last()	Push()	Enqueue()
Remove_First()	-	Dequeue()
Remove_Last()	Pop()	-