**STD::STRING (#include <string>)**

Access: char c = s2[1]; // 'e' (using array-like indexing)

String Size and Capacity

size\_t length = s2.size(); // or s2.length(); Returns the number of characters

bool isEmpty = s2.empty(); // Returns true if the string is empty

Appending and Insertion

s2 += " World"; // Appending using += operator

s2.append("!"); // Appending using `append` method

s2.insert(5, " dear"); // Inserting "dear" before the 6th character

Finding and Replacing

size\_t pos = s2.find("World"); // Returns starting position of the first match or

`std::string::npos` if not found

s2.replace(pos, 5, "Universe"); // Replaces "World" with "Universe"

Substrings

std::string subs = s2.substr(0, 5);

// Returns a substring starting from position 0 with length 5

**STD::VECTOR (#include <vector>)**

Initialization

std::vector<int> v1; // Default initialization

std::vector<int> v3 = {1, 2, 3, 4, 5}; // Initialization with initializer list

Capacity and Size Management

size\_t size = v1.size(); // Returns the number of elements

bool isEmpty = v1.empty(); // Returns true if the vector is empty

Modifying Elements

v1.push\_back(42); // Appends an integer with the value 42

v1.emplace\_back(42); // Constructs an element in-place at the end

v1.pop\_back(); // Removes the last element

v1.insert(v1.begin(), 0); // Inserts 0 at the beginning

v1.erase(v1.begin()); // Erases the first element

v1.clear(); // Removes all elements

Accessing Elements

int last = v2.back(); // Access the last element

int firstAgain = v2.front(); // Access the first element

Other Useful Operations

bool areEqual = (v2 == v3); // Compares if two vectors have the same elements in

the same order

for(const auto& elem : v2) { // python style for loop

// Use elem

}

| **SWITCH SYNTAX**  switch(variable) {  case value1:  // code to run if variable == value1  break;  case value2:  // code to run if variable == value2  break;  default:  // code to run if no case matches  } | **EXCEPTIONS**  try {  throw  } catch (const char\* e) {  std::cerr << "Error: " << e << std::endl;  } |
| --- | --- |

**ASCII**

In the ASCII standard, every character is assigned a unique 7-bit integer between 0 and 127.

For instance, the ASCII value of 'A' is 65, and the ASCII value of 'a' is 97.

Get ASCII value of a character:

char ch = 'A';

int asciiValue = static\_cast<int>(ch);

std::cout << "ASCII value of " << ch << ": " << asciiValue << std::endl;

// Outputs: ASCII value of A: 65

Convert ASCII value to a character:

int asciiVal = 97;

char character = static\_cast<char>(asciiVal);

std::cout << "Character for ASCII " << asciiVal << ": " << character << std::endl;

// Outputs: Character for ASCII 97: a

Converting lowercase to uppercase (and vice versa) by leveraging their ASCII difference:

char lower = 'c';

char upper = lower - ('a' - 'A');

std::cout << "Uppercase of " << lower << " is " << upper << std::endl;

// Outputs: Uppercase of c is C

**ISS OSS**

In C++, the header <sstream> provides string stream classes that facilitate in-memory string parsing and formatting operations. Essentially, they allow strings to be treated as streams, enabling you to perform input/output operations on them as you would with files.

| *std::istringstream (Input String Stream)*  Initialization:  std::string str = "123 456";  std::istringstream iss(str);  Reading from istringstream:  int a, b;  iss >> a >> b; // a will be 123 and b will be 456  Parsing a line of mixed types:  std::string mixed = "John 25";  std::istringstream mixstream(mixed);  std::string name;  int age;  mixstream >> name >> age; // name will be "John", age will be 25  ***FSTREAM #include <fstream>***  std::ifstream input("input.txt");  std::string line;  while (std::getline(input, line)) {// gets each line of file} | *std::ostringstream (Output String Stream)*  Initialization: std::ostringstream oss;  Writing to ostringstream:  oss << "Hello, " << "World!" << 123;  std::string output = oss.str(); // output will be "Hello, World!123"  Formatting strings:  int x = 42;  double y = 3.14;  oss << "int: " << x << ", double: " << y;  std::string formatted = oss.str(); // formatted will be "int: 42, double: 3.14"  *3. Common Member Functions*  .str(): Returns the contents of the string stream as a string.  .str(const std::string &s): Sets the contents of the string stream to the given string.  .clear(): Clears the string stream's error flags, it doesn’t clear the content. |
| --- | --- |

**LIMITS -** std::numeric\_limits<int>::max() std::numeric\_limits<int>::min()

**TYPE CASTING**

| C-style Cast:  double d = 10.5;  int i = (int)d; // d is explicitly converted to int  C++ Casts:  double d = 10.5;  int i = static\_cast<int>(d); | std::stoi, std::stod, ...: Convert string to int, double, etc.  std::string s = "123";  int i = std::stoi(s); // i = 123 as an integer  std::to\_string: Convert fundamental types to strings.  int i = 123;  std::string s = std::to\_string(i); |
| --- | --- |

**PARSING NUMBERS**

12345 % 10 gives 5 (the last digit)

12345 / 10 gives 1234 (the number without its last digit)

**POINTERS**

*1. Declaring Pointers*: int \*ptr is the same as int\* ptr

*2. Initializing Pointers* int x = 10; int \*ptr = &x;

*3. Accessing Value Using Pointers*

The \* operator is a dereference operator, and it's used to access the value at the address held by the pointer.

int x = 10;

int \*ptr = &x;

std::cout << \*ptr; // Outputs: 10

*4. Null Pointer*

int \*ptr = nullptr; // Using C++11 and later

*5. Dynamic Memory Allocation*

Pointers can be used with C++'s dynamic memory allocation functions: new and delete.

int \*ptr = new int; // Allocate memory for an integer

\*ptr = 10; // Assign value

delete ptr; // Free the allocated memory

*6. Arrays and Pointers*

Arrays in C++ are closely related to pointers. The name of the array is a pointer to the first element.

int arr[3] = {1, 2, 3};

int \*ptr = arr; // Pointing to arr[0]

**DYNAMIC MEMORY**

| **Allocation:**  new: Allocates memory for a single object. Returns a pointer to the object's type.  int\* ptr = new int(5); // Allocates an integer initialized with 5  new[]: Allocates memory for an array.  int\* arr = new int[10]; // Allocates an array of 10 integers | **Deallocation:**  delete: Deallocates memory allocated by new.  delete ptr;  delete[]: Deallocates memory allocated by new[].  delete[] arr; |
| --- | --- |

**DYNAMIC 1D AND 2D ARRAYS**

| Dynamic arrays are arrays whose size can be determined during runtime, as opposed to compile-time.  **Size Variable:** Always keep an auxiliary variable to track the size of the dynamic array since pointers don't store this information.  int\* dynamic1DArray; **// declare**  dynamic1DArray = new int[10]; **// allocate**  delete[] dynamic1DArray; **// deallocate** | int\*\* dynamic2DArray; **// declare**  dynamic2DArray = new int\*[5]; **// allocate**  for (int i = 0; i < 5; i++) {  dynamic2DArray[i] = new int[5];  }  for (int i = 0; i < 5; i++) { **// deallocate**  delete[] dynamic2DArray[i];  }  delete[] dynamic2DArray; |
| --- | --- |

**LINKED LIST**

| *Assume following structure:*  struct Node {  int value;  Node\* next;  Node(int num = 0) : value{num}, next{nullptr} {}  };  class LinkedList {  Node\* head;  public:  LinkedList() : head{nullptr} {}  // other member functions  }; | *Traversal:*  Node\* current = head;  while (current != nullptr) {  // do something  // current = current->next;  } |
| --- | --- |

**OOP SHIT**

# ifndef LINKEDLIST

# define LINKEDLIST

# include <iostream>

# include <string>

# include "Node.h"

class LinkedList {

private:

Node\* head;

Node\* tail;

public:

LinkedList(); // default constructor

~LinkedList(); // destructor

LinkedList(const LinkedList& other); // copy constructor

LinkedList& operator=(const LinkedList& other); // copy assignment

void insert(std::string location, int year, int month, double temperature); // insert a record to the linked list

void clear(); // clear the content of this linked list

std::string print() const;

Node\* getHead() const;

};

std::ostream& operator<<(std::ostream& os, const LinkedList& ll);

# endif

**INHERITANCE AND POLYMORPHISM**

| Inheritance:  class Animal {  public:  void eat() { /\* common function \*/ }  };  class Dog : public Animal {  public:  void bark() { /\* specific function \*/ }  }; | Polymorphism:  class Shape {  public:  virtual void draw() = 0; // Pure virtual function  };  class Circle : public Shape {  public:  void draw() override { /\* implementation \*/ }  };  class Rectangle : public Shape {  public:  void draw() override { /\* implementation \*/ }  }; |
| --- | --- |

**Protected keyword:** Members are accessible within the class and by derived class instances.

**Virtual Functions:** means can be overridden

**Abstract Classes (Pure Virtual):** (virtual void functionName() = 0;). These classes cannot be instantiated.

**Pure Virtual Functions:** A method that must be overridden in any non-abstract derived class.

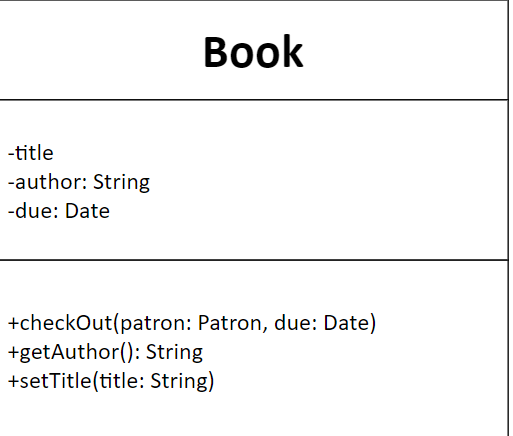
**When to Use Virtual and Pure Virtual Functions:**

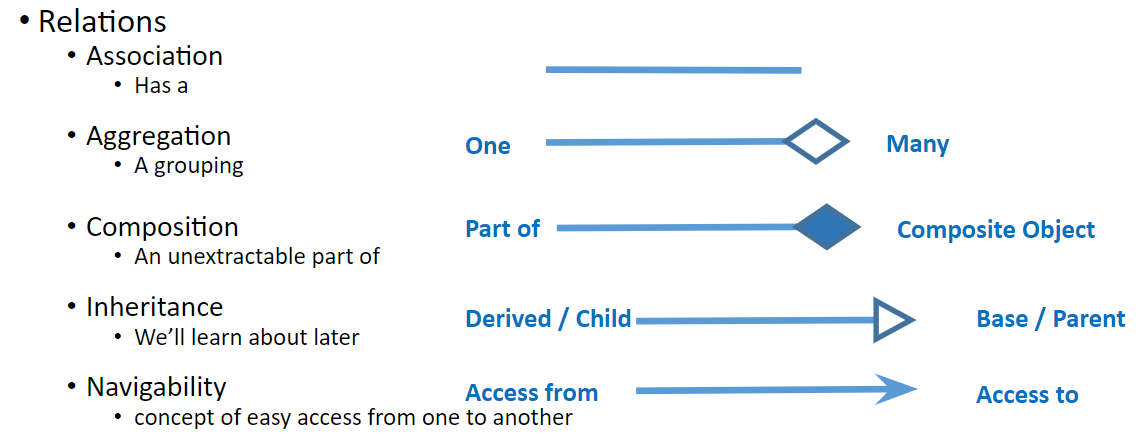
Use Virtual Functions when you want to allow a member function to be overridden in derived classes.

Use Pure Virtual Functions when you want to force derived classes to provide their own implementation of a function.

**If we need a collection of abstract classes:** We cannot create an instance of an abstract class.However, we can have a pointer to an abstract class - vector<BaseClass\*> v;

**UML CLASS DIAGRAMS**

- is private + is public



#ifndef MYSTRING\_H

#define MYSTRING\_H

#include <iostream>

#include <limits>

class MyString {

private:

unsigned int \_size;

unsigned int \_capacity;

char\* chars;

public:

size\_t npos = static\_cast<size\_t>(-1);

MyString(); // default

MyString(const MyString& mystr); // copy

MyString(const char\* s); // from c-string

~MyString();

void resize(size\_t n);

size\_t capacity() const;

size\_t size() const;

size\_t length() const;

const char\* data() const;

bool empty() const;

const char& front() const;

const char& at(size\_t pos) const;

void clear();

friend std::ostream& operator<< (std::ostream& os, const MyString& mystr);

MyString& operator= (const MyString& str);

MyString& operator= (const char\* s);

MyString& operator= (const char c);

MyString& operator+= (const MyString& str);

MyString& operator+= (const char\* s);

MyString& operator+= (const char c);

size\_t find(const MyString& mystr, size\_t pos = 0) const;

size\_t find(const char\* s, size\_t pos = 0) const;

size\_t find(const char\* s, size\_t pos, size\_t n) const;

size\_t find(const char c, size\_t pos = 0) const;

};

bool operator== (const MyString& lhs, const MyString& rhs);

bool operator== (const char\* lhs, const MyString& rhs);

bool operator== (const MyString& lhs, const char\* rhs);

MyString operator+ (const MyString& lhs, const MyString& rhs);

#endif

# include <iostream>

# include <string>

# include "LinkedList.h"

# include "Node.h"

using std::string, std::ostream;

LinkedList::LinkedList() : head(nullptr), tail(nullptr) {}

LinkedList::~LinkedList() {

Node\* current = head;

while (current != nullptr) {

Node\* next = current->next;

delete current;

current = next;

}

head = nullptr;

tail = nullptr;

}

LinkedList::LinkedList(const LinkedList& source) : head(nullptr), tail(nullptr) {

Node\* current = source.head;

while (current != nullptr) {

insert(current->data.id, current->data.year, current->data.month, current->data.temperature);

current = current->next;

}

}

LinkedList& LinkedList::operator=(const LinkedList& source) {

if (this == &source) {

return \*this;

}

this->clear();

Node\* current = source.head;

while (current != nullptr) {

this->insert(current->data.id, current->data.year, current->data.month, current->data.temperature);

current = current->next;

}

return \*this;

}

void LinkedList::insert(string location, int year, int month, double temperature) {

Node\* newNode = new Node(location, year, month, temperature);

if (head == nullptr || \*newNode < \*head) {

newNode->next = head;

head = newNode;

if (tail == nullptr) {

tail = newNode;

}

} else {

Node\* current = head;

while (current->next != nullptr && !(\*newNode < \*current->next)) {

current = current->next;

}

newNode->next = current->next;

current->next = newNode;

if (tail == current) {

tail = newNode;

}

}

}

void LinkedList::clear() {

Node\* current = head;

while (current != nullptr) {

Node\* next = current->next;

delete current;

current = next;

}

head = nullptr;

tail = nullptr;

}

Node\* LinkedList::getHead() const {

return head;

}

string LinkedList::print() const {

string outputString;

Node\* current = head;

while (current != nullptr) {

outputString += current->data.id + " "

+ std::to\_string(current->data.year) + " "

+ std::to\_string(current->data.month) + " ";

string tempStr = std::to\_string(current->data.temperature);

size\_t decimalPos = tempStr.find('.');

if (decimalPos != string::npos) {

size\_t end = tempStr.length() - 1;

while (end > decimalPos && tempStr[end] == '0') {

end--;

}

if (end == decimalPos) {

end--;

}

tempStr = tempStr.substr(0, end + 1);

}

outputString += tempStr + "\n";

current = current->next;

}

return outputString;

}

ostream& operator<<(ostream& os, const LinkedList& ll) {

os << ll.print();

return os;

}