|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| **S. No.** | **Name Of Experiment** | **Date** | **Pg. No.** | **Remarks** |
|  |  |  |  |  |

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| **S. No.** | **Name Of Experiment** | **Date** | **Pg. No.** | **Remarks** |
|  |  |  |  |  |

**Program No 1: Multiplication of two matrices using OOP**

**THEORY**

**MULTIDIMENSIONAL ARRAYS**

A multi-dimensional array can be termed as an array of arrays that stores homogeneous data in tabular form. Data in multidimensional arrays are stored in row-major order.

The***general form of declaring N-dimensional arrays*** is:

data\_type array\_name[size1][size2]....[sizeN];

* **data\_type**: Type of data to be stored in the array.
* **array\_name**: Name of the array
* **size1, size2,… ,sizeN**: Sizes of the dimension

**TWO DIMENSIONAL ARRAYS (MATRIX)**

Two – dimensional array is the simplest form of a multidimensional array. We can see a two – dimensional array as an array of one-dimensional array for easier understanding.

The basic form of declaring a two-dimensional array of size x, y:   
**Syntax:**

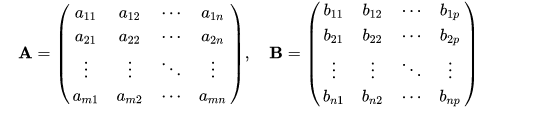
**data\_type array\_name[x][y];**

We can declare a two-dimensional integer array say ‘x’ of size 10,20 as:

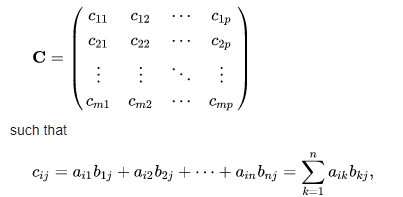
int x[10][20];

**MATRIX MULTIPLICATION**

If **A** is an *m* × *n* matrix and **B** is an *n* × *p* matrix,



the *matrix product* **C** = **AB** (denoted without multiplication signs or dots) is defined to be the *m* × *p* matrix



for *i* = 1, ..., *m* and *j* = 1, ..., *p*.

**SOURCE CODE**

// Header Files

#include <iostream>

// Prototyping the functions

int \*input2D\_matrix1D(int \*, int \*);

int \*AxB(int \*, int \*, int, int, int);

void print2Dmatrix(int \*, int, int);

int main()

{

    // Getting the two matrices A and B inputted

    // A1D namins is used becuase its a 2D array A but in 1D form

    int ARows, ACols, BRows, BCols;

    std::cout << "Input data for Matrix A:" << std::endl;

    int \*A1D = input2D\_matrix1D(&ARows, &ACols);

    std::cout << "\nInput data for Matrix B:" << std::endl;

    int \*B1D = input2D\_matrix1D(&BRows, &BCols);

    // Checking if the matrices are valid for being multiplied

    if (ACols != BRows)

    {

        std::cout << "Inputted matrix cant be multiplied.";

        return 0;

    }

    // Multiplication using function

    int \*C1D = AxB(A1D, B1D, ARows, ACols, BCols);

    // Printing using function

    print2Dmatrix(C1D, ARows, BCols);

    return 0;

}

int \*AxB(int \*A, int \*B, int ARows, int ACols, int BCols)

{

    // Creating a 2D array with size ARows X BCols

    int(\*C)[ARows] = (int(\*)[ARows]) new int[ARows \* BCols];

    // Matrix multiplication logic

    for (int i = 0; i < ARows; i++)

        for (int j = 0; j < BCols; j++)

        {

            C[i][j] = 0;

            for (int k = 0; k < ACols; k++)

                C[i][j] += \*(A + ACols \* i + k) \* \*(B + ACols \* k + j);

        }

    return (int \*)C;

}

int \*input2D\_matrix1D(int \*rows, int \*columns)

{

    // This function takes input from the user and creates an array from that

    printf("Enter the number of Rows: ");

    scanf("%d", rows); // It also returns the value back because pointer

    printf("Enter the number of Columns: ");

    scanf("%d", columns);

    // Creates an array with length as if the 2D matrix was converted to 1D row wise

    // This is done since you cannot pass in function and return a variable length multidimensional array

    int \*matrix = new int[(\*rows) \* (\*columns)];

    for (int i = 0; i < \*rows; i++)

        for (int j = 0; j < \*columns; j++)

        {

            printf("Enter the number at position (%d,%d): ", i, j);

            scanf("%d", matrix + i \* (\*columns) + j);

        }

    return matrix;

}

void print2Dmatrix(int \*A1D, int rows, int columns)

{

    // Converting 1D array into 2D array using type casting

    int(\*A)[rows] = (int(\*)[rows])A1D;

    std::cout << "\nMatrix C = AxB:\n";

    for (int i = 0; i < rows; i++)

    {

        for (int j = 0; j < columns; j++)

            std::cout << "\t" << A[i][j];

        std::cout << "\n";

    }

}

**OUTPUT**

* Test case 1: Where matrices can be multiplied, ie: Columns of A = Rows of B

PS C:\Users\admin\Documents\Saksham Gupta\MSIT> ./a.exe

Input data for Matrix A:

Enter the number of Rows: 2

Enter the number of Columns: 3

Enter the number at position (0,0): 2

Enter the number at position (0,1): 4

Enter the number at position (0,2): 1

Enter the number at position (1,0): 2

Enter the number at position (1,1): 5

Enter the number at position (1,2): 3

Input data for Matrix B:

Enter the number of Rows: 3

Enter the number of Columns: 1

Enter the number at position (0,0): 5

Enter the number at position (1,0): 6

Enter the number at position (2,0): 3

Matrix C = AxB:

20

70

* Test case 2: Where matrices can’t be multiplied, ie: Columns of A != Rows of B

PS C:\Users\admin\Documents\Saksham Gupta\MSIT> ./a.exe

Input data for Matrix A:

Enter the number of Rows: 1

Enter the number of Columns: 2

Enter the number at position (0,0): 4

Enter the number at position (0,1): 5

Input data for Matrix B:

Enter the number of Rows: 4

Enter the number of Columns: 1

Enter the number at position (0,0): 2

Enter the number at position (1,0): 3

Enter the number at position (2,0): 4

Enter the number at position (3,0): 5

Inputted matrix cant be multiplied.

**Program No 2: Write a program to perform addition of two complex numbers using constructor overloading. The first constructor which takes no argument is used to create objects which are not initialized, second which takes one argument is used to initialize real and imag parts to equal values and third which takes two argument is used to initialized real and imag to two different values.**

**THEORY**

**CONSTRUCTOR**

**Constructor in C++** is a special method that is invoked automatically at the time of object creation. It is used to initialize the data members of new objects generally. The constructor in C++ has the same name as the class or structure. Constructor is invoked at the time of object creation. It constructs the values i.e. provides data for the object which is why it is known as constructors.

Constructor does not have a return value, hence they do not have a return type.

The prototype of Constructors is as follows:

**<class-name> (list-of-parameters);**

Constructors can be defined inside or outside the class declaration:

**CONSTRUCTOR OVERLOADING**

In C++, We can have more than one constructor in a class with same name, as long as each has a different list of arguments.This concept is known as Constructor Overloading and is quite similar to function overloading.

* Overloaded constructors essentially have the same name (exact name of the class) and different by number and type of arguments.
* A constructor is called depending upon the number and type of arguments passed.
* While creating the object, arguments must be passed to let compiler know, which constructor needs to be called.

**SOURCE CODE**

#include <iostream> // Header files

class Complex

{

// The class for Complex numbers

private:

float real\_part;

float imag\_part;

public:

// Constructor with no arguments, sets both values as 0

Complex()

{

this->real\_part = 0;

this->imag\_part = 0;

}

// Constructor with 1 argument, sets both values as one

Complex(float value)

{

this->real\_part = value;

this->imag\_part = value;

}

// Constructor with 2 arguments, 1 for each part

Complex(float real\_part, float imag\_part)

{

this->real\_part = real\_part;

this->imag\_part = imag\_part;

}

// + Operator overloading for addition

Complex operator+(const Complex &other)

{

return Complex(this->real\_part + other.real\_part, this->imag\_part + other.imag\_part);

}

// ostream operator overloading for printing

friend std::ostream &operator<<(std::ostream &output, const Complex &number)

{

output << number.real\_part << " + (" << number.imag\_part << ")i";

return output;

}

};

// Driver code

int main()

{

// Initialising 3 Complex numbers using each constructor

Complex A;

Complex B(2.6);

Complex C(1.4, 4.5);

// Printing them and their addition

std::cout << "A = " << A << ", B = " << B << ", C = " << C << std::endl;

std::cout << "B+C = " << B + C;

return 0;

}

**OUTPUT**

PS C:\Users\admin\Documents\Saksham Gupta\MSIT> ./a.exe

A = 0 + (0)i, B = 2.6 + (2.6)i, C = 1.4 + (4.5)i

B+C = 4 + (7.1)i

**Program No 3: Write a program to find the greatest of two given numbers in two classes using friend function**

**THEORY**

**FRIEND FUNCTION**

Friend Function Like friend class, a friend function can be given a special grant to access private and protected members. A friend function can be:

a) A member of another class

b) A global function

Features of Friend function:

* A friend function is a special function in C++ which in-spite of not being member function of a class has privilege to access private and protected data of a class.
* A friend function is a non member function or ordinary function of a class, which is declared as a friend using the keyword “friend” inside the class. By declaring a function as a friend, all the access permissions are given to the function.
* The keyword “friend” is placed only in the function declaration of the friend function and not in the function definition.
* When friend function is called neither name of object nor dot operator is used. However it may accept the object as argument whose value it want to access.
* Friend function can be declared in any section of the class i.e. public or private or protected.

**SOURCE CODE**

#include <iostream> // Header files

#include <string>

class BankAccount

{

// The class for Bank Account

private:

std::string name;

float balance;

public:

// Constructor

BankAccount(std::string name, float balance)

{

this->name = name;

this->balance = balance;

}

// ostream operator overloading for printing

friend std::ostream &operator<<(std::ostream &output, const BankAccount &acc)

{

output << "Name: " << acc.name << ", Balance: " << acc.balance;

return output;

}

// friend function for printing greater balance

friend void greater\_balance(const BankAccount &, const BankAccount &);

};

// Friend function definiton

void greater\_balance(const BankAccount &acc1, const BankAccount &acc2)

{

std::cout << (acc1.balance > acc2.balance ? acc1.name : acc2.name) << " has more balance";

}

// Driver code

int main()

{

// Initialising few objects

BankAccount acc1("Saksham Gupta", 1000000);

BankAccount acc2("Jatin Suteri", 0.02);

// Printing them

std::cout << acc1 << std::endl;

std::cout << acc2 << std::endl;

// Calling the friend function

greater\_balance(acc1, acc2);

}

**OUTPUT**

PS C:\Users\admin\Documents\Saksham Gupta\MSIT> ./a.exe

Name: Saksham Gupta, Balance: 1e+06

Name: Jatin Suteri, Balance: 0.02

Saksham Gupta has more balance

**Program No 4: Implement a class string containing the following functions:**

1. **Overload + operator to carry out the concatenation of strings.**
2. **Overload = operator to carry out string copy.**
3. **Overload <= operator to carry out the comparison of strings.**
4. **Function to display the length of a string.**
5. **Function tolower( ) to convert upper case letters to lower case.**
6. **Function toupper( ) to convert lower case letters to upper case.**

**THEORY**

**OPERATOR OVERLOADING**

In C++, we can make operators work for user-defined classes. This means C++ has the ability to provide the operators with a special meaning for a data type, this ability is known as operator overloading. For example, we can overload an operator ‘+’ in a class like String so that we can concatenate two strings by just using +. Other example classes where arithmetic operators may be overloaded are Complex Numbers, Fractional Numbers, Big Integer, etc.

Operator overloading is a compile-time polymorphism. It is an idea of giving special meaning to an existing operator in C++ without changing its original meaning.

**SOURCE CODE**

#include <iostream> // Header files

class String

{

// The class for user made Character String

private:

char \*\_value;

int \_length = 0;

// Constructor with direct value and length

String(char \*value, int length)

{

this->\_value = value;

this->\_length = length;

}

public:

// Constructor

String(const char \*value)

{

while (value[this->\_length] != '\0')

this->\_length++;

this->\_value = new char[this->\_length + 1];

for (int i = 0; i <= this->\_length; i++)

{

this->\_value[i] = value[i];

}

}

// Default Constructor

String()

{

this->\_length = 0;

char \*value = new char[1];

value[0] = '\0';

this->\_value = value;

}

// + Operator overloading for Concatenation

String operator+(const String &other)

{

int new\_len = this->\_length + other.\_length;

char \*value = new char[new\_len + 1];

for (int i = 0; i < this->\_length; i++)

{

value[i] = this->\_value[i];

}

for (int i = 0; i <= other.\_length; i++)

{

value[this->\_length + i] = other.\_value[i];

}

return String(value, new\_len);

}

// = Operator overloading for copying

void operator=(const String &other)

{

this->\_length = other.\_length;

char \*value = new char[this->\_length + 1];

for (int i = 0; i <= this->\_length; i++)

{

value[i] = other.\_value[i];

}

free(this->\_value);

this->\_value = value;

}

// Copy constructor

String(const String &other)

{

this->\_value = NULL;

this->operator=(other);

}

// <= Operator overloading for comparison

bool operator<=(const String &other)

{

for (int i = 0; i <= this->\_length; i++)

{

if (this->\_value[i] != other.\_value[i])

{

if (this->\_value[i] < other.\_value[i])

return true;

else

return false;

}

}

return true;

}

// returning the length of the string

int length()

{

return \_length;

}

// converting the string to upper case

void toUpper()

{

for (int i = 0; i < this->\_length; i++)

{

this->\_value[i] = toupper(this->\_value[i]);

}

}

// Converting the string to lower case

void toLower()

{

for (int i = 0; i < this->\_length; i++)

{

this->\_value[i] = tolower(this->\_value[i]);

}

}

// ostream operator overloading for printing

friend std::ostream &operator<<(std::ostream &output, const String &string)

{

output << string.\_value;

return output;

}

};

// Driver code

int main()

{

String s1("Saksham");

String s2("Gupta");

// Using of concatenation and copy

String s3 = s1 + s2;

// Using of copy

String s4 = s1;

std::cout << "String 1 = " << s1 << ", String 2 = " << s2 << std::endl;

std::cout << "Concatenation of String 1 and 2 (String 3) = " << s3 << std::endl;

std::cout << "Copy of String 1 (String 4) = " << s4 << std::endl;

std::cout << "Comparison (String 1 <= String 3) = " << (s1 <= s3) << std::endl; // less than

std::cout << "Comparison (String 1 <= String 2) = " << (s1 <= s2) << std::endl; // greater than

std::cout << "Comparison (String 1 <= String 1) = " << (s1 <= s1) << std::endl; // equal

std::cout << "Length of String 1 = " << s1.length() << std::endl;

s1.toLower();

std::cout << "String 1 toLower() = " << s1 << std::endl;

s1.toUpper();

std::cout << "String 1 toUpper() = " << s1 << std::endl;

std::cout << "Copy didn't change (String 4) = " << s4 << std::endl;

return 0;

}

**OUTPUT**

PS C:\Users\admin\Documents\Saksham Gupta\MSIT> ./a.exe

String 1 = Saksham, String 2 = Gupta

Concatenation of String 1 and 2 (String 3) = SakshamGupta

Copy of String 1 (String 4) = Saksham

Comparison (String 1 <= String 3) = 1

Comparison (String 1 <= String 2) = 0

Comparison (String 1 <= String 1) = 1

Length of String 1 = 7

String 1 toLower() = saksham

String 1 toUpper() = SAKSHAM

Copy didn't change (String 4) = Saksham

**Program No 5: Create a class called LIST with two pure virtual function store() and retrieve().To store a value call store and to retrieve call retrieve function. Derive two classes stack and queue from it and override store and retrieve.**

**THEORY**

**VIRTUAL FUNCTIONS**

A virtual function is a member function which is declared within a base class and is re-defined (overridden) by a derived class. When you refer to a derived class object using a pointer or a reference to the base class, you can call a virtual function for that object and execute the derived class’s version of the function.

* Virtual functions ensure that the correct function is called for an object, regardless of the type of reference (or pointer) used for function call.
* They are mainly used to achieve Runtime polymorphism
* Functions are declared with a **virtual** keyword in base class.
* The resolving of function call is done at runtime.

**Rules for Virtual Functions**

1. Virtual functions cannot be static.
2. A virtual function can be a friend function of another class.
3. Virtual functions should be accessed using pointer or reference of base class type to achieve runtime polymorphism.
4. The prototype of virtual functions should be the same in the base as well as derived class.
5. They are always defined in the base class and overridden in a derived class. It is not mandatory for the derived class to override (or re-define the virtual function), in that case, the base class version of the function is used.
6. A class may have virtual destructor but it cannot have a virtual constructor.

A pure virtual function (or abstract function) in C++ is a virtual function for which we can have implementation, But we must override that function in the derived class, otherwise the derived class will also become abstract class

A pure virtual function is declared by assigning 0 in declaration.

**SOURCE CODE**

#include <iostream> // Header Files

// Node struct as data element of a list/stack/queue, using concept of linked list

struct Node

{

int data;

Node \*next;

};

// Parent Class list

class List

{

protected:

// Protected to be inherited

Node \*head = NULL;

Node \*tail = NULL;

public:

// Virtual for overriding in derived class

virtual void store(int n) = 0; // Pure virtual functions since assigned to 0

virtual int retrive() = 0;

};

// Derived Class Stack from Class List

class Stack : public List

{

public:

// Overriding the virtual functions of class List

void store(int num)

{

// Stack Push Logic

Node \*n1 = new Node;

n1->data = num;

n1->next = NULL;

if ((this->head == NULL) && (this->tail == NULL))

{

this->head = n1;

this->tail = n1;

}

else

{

this->tail->next = n1;

this->tail = n1;

}

}

int retrive()

{

// Stack Pop Logic

if ((this->tail == NULL) && (this->head == NULL))

{

return -1;

}

else

{

int n = this->tail->data;

Node \*ptr = this->head;

while ((ptr->next != this->tail) && (this->head != this->tail))

{

ptr = ptr->next;

}

ptr->next = NULL;

free(this->tail);

if (this->head != this->tail)

{

this->tail = ptr;

}

else

{

this->tail = NULL;

this->head = NULL;

}

return n;

}

}

// ostream operator overloading for printing

friend std::ostream &operator<<(std::ostream &output, const Stack &stack)

{

// Printing by Linked list traversing

Node \*ptr = stack.head;

while (ptr != NULL)

{

output << ptr->data << std::endl;

ptr = ptr->next;

}

return output;

}

};

// Derived Class Queue from base class List

class Queue : public List

{

public:

// Overriding the virtual functions of class List

void store(int n)

{

// Queue Insert Logic

Node \*n1 = new Node;

n1->data = n;

n1->next = NULL;

if ((this->head == NULL) && (this->tail == NULL))

{

this->head = n1;

this->tail = n1;

}

else

{

this->tail->next = n1;

this->tail = n1;

}

}

int retrive()

{

// Queue Delete Logic

if ((this->tail == NULL) && (this->head == NULL))

{

return -1;

}

else

{

int n = this->head->data;

if (this->head == this->tail)

{

this->head = this->tail = NULL;

}

else

{

this->head = this->head->next;

}

return n;

}

}

// ostream operator overloading for printing

friend std::ostream &operator<<(std::ostream &output, const Queue &queue)

{

// Printing by Linked list traversing

Node \*ptr = queue.head;

while (ptr != NULL)

{

output << ptr->data << std::endl;

ptr = ptr->next;

}

return output;

}

};

// Driver Code

int main()

{

// Making objects of Stack and Queue

Stack S;

Queue Q;

// Performing operations on Stack

std::cout << "Stack: " << std::endl;

std::cout << S;

S.store(10);

S.store(20);

std::cout << "After Storing twice Stack: " << std::endl;

std::cout << S;

std::cout << "Retriving once: " << S.retrive() << std::endl;

std::cout << "Retriving twice: " << S.retrive() << std::endl;

std::cout << "Retriving thrice: " << S.retrive() << std::endl

<< std::endl;

// Performing operations on Queue

std::cout << "Queue: " << std::endl;

std::cout << Q;

Q.store(30);

Q.store(40);

std::cout << "After Storing twice Queue: " << std::endl;

std::cout << Q;

std::cout << "Retriving once: " << Q.retrive() << std::endl;

std::cout << "Retriving twice: " << Q.retrive() << std::endl;

std::cout << "Retriving thrice: " << Q.retrive() << std::endl;

return 0;

}

**OUTPUT**

PS C:\Users\admin\Documents\Saksham Gupta\MSIT> ./a.exe

Stack:

After Storing twice Stack:

10

20

Retriving once: 20

Retriving twice: 10

Retriving thrice: -1

Queue:

After Storing twice Queue:

30

40

Retriving once: 30

Retriving twice: 40

Retriving thrice: -1

**Program No 6: Write a program to define the function template for calculating the square of given numbers with different data types.**

**THEORY**

**FUNCTION TEMPLATE**

Function templates are special functions that can operate with generic types. This allows us to create a function template whose functionality can be adapted to more than one type or class without repeating the entire code for each type.

In C++ this can be achieved using template parameters. A template parameter is a special kind of parameter that can be used to pass a type as argument: just like regular function parameters can be used to pass values to a function, template parameters allow to pass also types to a function. These function templates can use these parameters as if they were any other regular type.

The format for declaring function templates with type parameters is:

**template <class identifier> function\_declaration;**

**template <typename identifier> function\_declaration;**

**SOURCE CODE**

#include <iostream> // Header Files

using namespace std;

// Template declaration

template <class number>

// Template function

number square(number num)

{

return num \* num;

}

// Driver Code

int main()

{

int int\_num;

float float\_num;

cout << "Enter a integer number:\t";

cin >> int\_num;

cout << "Squared integer number:\t" << square(int\_num) << endl;

cout << "Enter a floating-point number:\t";

cin >> float\_num;

cout << "Squared floating-point number:\t" << square(float\_num) << endl;

}

**OUTPUT**

PS C:\Users\admin\Documents\Saksham Gupta\MSIT> ./a.exe

Enter a integer number: 24

Squared integer number: 576

Enter a floating-point number: 19.26

Squared floating-point number: 370.948

**Program No 7: Write a program to demonstrate the use of special functions, constructor and destructor in the class template. The program is used to find the bigger of two entered numbers.**

**THEORY**

**CLASS TEMPLATE**

A class template starts with the keyword template followed by template parameter(s) inside <> which is followed by the class declaration.

**template <class T>**

**class className {**

**private:**

**T var;**

**... .. ...**

**public:**

**T functionName(T arg);**

**... .. ...**

**};**

In the above declaration, T is the template argument which is a placeholder for the data type used, and class is a keyword.

**SPECIAL MEMBER FUNCTIONS**

The *special member functions* are class (or struct) member functions that, in certain cases, the compiler automatically generates for you. These functions are the default constructor, the destructor, the copy constructor and copy assignment operator, and the move constructor and move assignment operator. If your class does not define one or more of the special member functions, then the compiler may implicitly declare and define the functions that are used.

**DEFAULT CONSTRUCTOR**

*Default constructors* typically have no parameters, but they can have parameters with default values.

class Box {

public:

Box() { /\*perform any required default initialization steps\*/}

// All params have default values

Box (int w = 1, int l = 1, int h = 1): m\_width(w), m\_height(h), m\_length(l){}

...

}

**DESTRUCTOR**

A destructor is a member function that is invoked automatically when the object goes out of scope or is explicitly destroyed by a call to delete. A destructor has the same name as the class, preceded by a tilde (~). For example, the destructor for class String is declared: **~String().**

**COPY CONSTRUCTOR**

In the [C++](https://en.wikipedia.org/wiki/C%2B%2B)[programming language](https://en.wikipedia.org/wiki/Programming_language), a copy constructor is a special [constructor](https://en.wikipedia.org/wiki/Constructor_(computer_science)) for creating a new [object](https://en.wikipedia.org/wiki/Object_(computer_science))[as a copy](https://en.wikipedia.org/wiki/Object_copy) of an existing object. Copy constructors are the standard way of copying objects in C++, as opposed to [cloning](https://en.wikipedia.org/wiki/Cloning_(programming)), and have C++-specific nuances.

The first argument of such a constructor is a reference to an object of the same type as is being constructed (const or non-const), which might be followed by parameters of any type (all having default values).

**COPY ASSIGNMENT**

The **copy assignment operator**, often just called the "assignment operator", is a special case of assignment operator where the source (right-hand side) and destination (left-hand side) are of the same class type. It is one of the special member functions, which means that a default version of it is generated automatically by the compiler if the programmer does not declare one. The default version performs a memberwise copy, where each member is copied by its own copy assignment operator (which may also be programmer-declared or compiler-generated).

**MOVE CONSTRUCTOR**

In C++03 (and before), temporaries (termed "rvalues", as they often lie on the right side of an assignment) were intended to never be modifiable — just as in C — and were considered to be indistinguishable from const T& types; nevertheless, in some cases, temporaries could have been modified, a behavior that was even considered to be a useful loophole. C++11 adds a new non-const reference type called an rvalue reference, identified by T&&. This refers to temporaries that are permitted to be modified after they are initialized, for the purpose of allowing "move semantics".

In C++11, a move constructor of std::vector<T> that takes an rvalue reference to an std::vector<T> can copy the pointer to the internal C-style array out of the rvalue into the new std::vector<T>, then set the pointer inside the rvalue to null. Since the temporary will never again be used, no code will try to access the null pointer, and because the pointer is null, its memory is not deleted when it goes out of scope. Hence, the operation not only forgoes the expense of a deep copy, but is safe and invisible.

**MOVE ASSIGNMENT**

In the C++ programming language, the **move assignment operator**= is used for transferring a temporary object to an existing object. The move assignment operator, like most C++ operators, can be overloaded. Like the copy assignment operator it is a special member function

|  |  |
| --- | --- |
| **Function** | **syntax for class MyClass** |
| Default constructor | MyClass(); |
| Copy constructor | MyClass(const MyClass& other); |
| Move constructor | MyClass(MyClass&& other) noexcept; |
| Copy assignment operator | MyClass& operator=(const MyClass& other); |
| Move assignment operator | MyClass& operator=(MyClass&& other) noexcept; |
| Destructor | ~MyClass(); |

.

**SOURCE CODE**

#include <iostream> // Header Files

// Template class declaration

template <class number>

class Number

{

private:

number \*\_value;

public:

// Constructor

Number(number value)

{

printf("Constructor Called!\n");

this->\_value = new number;

\*(this->\_value) = value;

}

// Default Constructor

Number()

{

printf("Default Constructor Called!\n");

this->\_value = new number;

\*(this->\_value) = 0;

}

// Copy Constructor

Number(const Number &other)

{

printf("Copy Constructor Called!\n");

this->\_value = new number;

\*(this->\_value) = \*(other.\_value);

}

// Copy Assignment Operator

Number &operator=(const Number &other)

{

printf("Copy Assignment Called!\n");

\*(this->\_value) = \*(other.\_value);

return \*this;

}

// Move Constructor

Number(Number &&other)

{

printf("Move Constructor Called!\n");

this->\_value = other.\_value;

other.\_value = nullptr;

}

// Move Assignment Operator

Number &operator=(Number &&other)

{

printf("Move Assignment Called!\n");

this->\_value = other.\_value;

other.\_value = nullptr;

return \*this;

}

// Destructor

~Number()

{

printf("Destructor Called!\n");

delete this->\_value;

}

// Function to read \_value

number value() { return \*(this->\_value); }

// Overloading the + operator for addtion used for showing move constructor

Number operator+(const Number &other)

{

return Number(\*(this->\_value) + \*(other.\_value));

}

// Overloading the > operator for finding greater of 2 Number classes

bool operator>(const Number &other)

{

return \*(this->\_value) > \*(other.\_value);

}

// ostream operator overloading for printing

friend std::ostream &operator<<(std::ostream &output, const Number &num)

{

output << \*(num.\_value);

return output;

}

};

// Driver Code

int main()

{

// Getting the initial 2 values to perform operations on

float f1, f2;

std::cout << "Enter the number 1: ";

std::cin >> f1;

std::cout << "Enter the number 2: ";

std::cin >> f2;

std::cout << std::endl;

std::cout << "Demonstration of various special member functions of class: \n";

std::cout << std::endl;

// Using floats to create the objects of Number

std::cout << "Objects A and B (basic constructors): \n";

Number<float> a(f1); // Constructor

Number<float> b(f2); // Constructor

std::cout << std::endl;

std::cout << "Object C (Defualt constructor): \n";

Number<float> c; // Default Constructor

std::cout << std::endl;

std::cout << "Object D (Move constructor but elision): \n";

Number<float> d = b + a; // Copy Elision

std::cout << std::endl;

std::cout << "Object C = B + A (Move assignment operator): \n";

c = b + a; // Move Assignment Operator

std::cout << std::endl;

std::cout << "Object E (Move constructor): \n";

Number<float> e = std::move(b + a); // Move Constructor

std::cout << std::endl;

std::cout << "Object F (Copy constructor): \n";

Number<float> f = b; // Copy constructor

std::cout << std::endl;

std::cout << "Object C = F (Copy Assignment): \n";

c = f; // Copy Assignment Operator

std::cout << std::endl;

// Checking if A>B

std::cout << "Checking which from A and B is bigger: \n"; // Overloaded > Operator

std::cout << (a > b ? a.value() : b.value()) << " is bigger\n";

std::cout << std::endl;

return 0;

}

**OUTPUT**

PS C:\Users\admin\Documents\Saksham Gupta\MSIT> ./a.exe

Enter the number 1: 10.2

Enter the number 2: 2

Demonstration of various special member functions of class:

Objects A and B (basic constructors):

Constructor Called!

Constructor Called!

Object C (Defualt constructor):

Default Constructor Called!

Object D (Move constructor but elision):

Constructor Called!

Object C = B + A (Move assignment operator):

Constructor Called!

Move Assignment Called!

Destructor Called!

Object E (Move constructor):

Constructor Called!

Move Constructor Called!

Destructor Called!

Object F (Copy constructor):

Copy Constructor Called!

Object C = F (Copy Assignment):

Copy Assignment Called!

Checking which from A and B is bigger:

10.2 is bigger

Destructor Called!

Destructor Called!

Destructor Called!

Destructor Called!

Destructor Called!

Destructor Called!

**Program No 8: Write a program to perform the deletion of white spaces such as horizontal tab, vertical tab, space, line feed, new line and carriage return from a text file and store the contents of the file without the white spaces on another file.**

**THEORY**

**FILE HANDLING**

### **Classes for File stream operations :-**

The I/O system of C++ contains a set of classes which define the file handling methods. These include ifstream, ofstream and fstream classes. These classes area derived from fstream and from the corresponding iostream class. These classes, designed to manage the disk files, are declared in fstream and therefore we must include this file in any program that uses files.  
1. ios:-

* ios stands for input output stream.
* This class is the base class for other classes in this class hierarchy.
* This class contains the necessary facilities that are used by all the other derived classes for input and output operations.

2. istream:-

* istream stands for input stream.
* This class is derived from the class ‘ios’.
* This class handle input stream.
* The extraction operator(>>) is overloaded in this class to handle input streams from files to the program execution.
* This class declares input functions such as get(), getline() and read().

3. ostream:-

* ostream stands for output stream.
* This class is derived from the class ‘ios’.
* This class handle output stream.
* The insertion operator(<<) is overloaded in this class to handle output streams to files from the program execution.
* This class declares output functions such as put() and write().

4. streambuf:-

* This class contains a pointer which points to the buffer which is used to manage the input and output streams.

5. fstreambase:-

* This class provides operations common to the file streams. Serves as a base for fstream, ifstream and ofstream class.
* This class contains open() and close() function.

6. ifstream:-

* This class provides input operations.
* It contains open() function with default input mode.
* Inherits the functions get(), getline(), read(), seekg() and tellg() functions from the istream.

7. ofstream:-

* This class provides output operations.
* It contains open() function with default output mode.
* Inherits the functions put(),  write(), seekp() and tellp() functions from the ostream.

8. fstream:-

* This class provides support for simultaneous input and output operations.
* Inherits all the functions from istream and ostream classes through iostream.

9. filebuf:-

* Its purpose is to set the file buffers to read and write.
* We can also use file buffer member function to determine the length of the file.

In C++, files are mainly dealt by using three classes fstream, ifstream, ofstream available in fstream headerfile.   
**ofstream**: Stream class to write on files   
**ifstream**: Stream class to read from files   
**fstream**: Stream class to both read and write from/to files.

**SOURCE CODE**

#include <iostream> // Header files

#include <fstream>

#include <string>

#include <algorithm>

// Declarations

std::ifstream InputFile;

std::ofstream OutputFile;

// WhiteSpace character set

char whitespaces[] = {' ', '\n', '\t', '\r', '\v'};

int whitespaces\_size = sizeof(whitespaces) / sizeof(\*whitespaces);

// Driver code

int main()

{

// Open the input and output files

InputFile.open("08\_whitespace.txt");

OutputFile.open("08\_shredded.txt");

// Cant open the input file

if (!InputFile)

{

std::cout << "Input file could not be opened! Terminating!" << std::endl;

return 1;

}

// Cant open the output file

if (!OutputFile)

{

std::cout << "Output file could not be opened! Terminating!" << std::endl;

return 1;

}

// a is the iterated character in inputfile

char a;

while (!InputFile.eof()) // Not end of file

{

InputFile.get(a);

// Check if a in whitespace character set

bool exists = std::find(whitespaces, whitespaces + whitespaces\_size, a) != whitespaces + whitespaces\_size;

// If a not whitespace write in output file

if (!exists)

OutputFile << a;

}

// Close the output file

OutputFile.close();

return 0;

}

**OUTPUT**

08\_whitespace.txt

What is Lorem Ipsum?

Lorem Ipsum is simply dummy text of the printing and

typesetting industry. Lorem Ipsum has been the

industry's standard dum my text ever sinc e the 1500s, when an unknown

printer took a galley of type and scrambled it to make a type specimen book. It has

survived not only five cen turies, but also the leap into electronic typ esetting, remaining e

ssentially unchanged. It was

popularised in the 1960s with the release of Letraset sheets containing Lorem Ipsum

passages, and more recently with desktop publishing software like Aldus PageMaker including versions of Lorem Ipsum.

Why do we use it?

08\_shredded.txt

WhatisLoremIpsum?LoremIpsumissimplydummytextoftheprintingandtypesettingindustry.LoremIpsumhasbeentheindustry'sstandarddummytexteversincethe1500s,whenanunknownprintertookagalleyoftypeandscrambledittomakeatypespecimenbook.Ithassurvivednotonlyfivecenturies,butalsotheleapintoelectronictypesetting,remainingessentiallyunchanged.Itwaspopularisedinthe1960swiththereleaseofLetrasetsheetscontainingLoremIpsumpassages,andmorerecentlywithdesktoppublishingsoftwarelikeAldusPageMakerincludingversionsofLoremIpsum.Whydoweuseit?

**Program No 9: Write a program to read the class object of student info such as name , age ,sex ,height and weight from the**

**keyboard and to store them on a specified file using read() and write() functions. Again the same file is**

**opened for reading and displaying the contents of the file on the screen.**

**THEORY**

**STORING OBJECTS INTO FILES**

The data transfer is usually done using '>>' and <<' operators. But if you have a class with 4 data members and want to write all 4 data members from its object directly to a file or vice-versa, we can do that using following syntax :

file1.write( (char\*)&Emp\_1, sizeof(Emp1) );

* Here data present in class object Emp\_1 is written to file Employee.txt by calling write function. (char\*)&Emp\_1 is used to point at the start of an object and sizeof(Emp\_1) calculates the number of bytes copied in file.

file2.read( (char\*)&Emp\_1, sizeof(Emp1) );

* Here data present in class object Emp\_1 is read from file Employee.txt by calling read function. (char\*)&Emp\_1 is used to point at the start of an object and sizeof(Emp\_1) calculates the number of bytes read from the file.

**SOURCE CODE**

#include <iostream> // Header Files

#include <fstream>

// Class Student

class Student

{

private:

char name[40];

int age;

float height;

float weight;

char sex;

public:

void putdata();

void getdata();

};

// Defining the function putdata() to enter the values of data members of an object.

void Student ::putdata()

{

std::cout << "Enter the name : ";

std::cin.getline(this->name, 40);

std::cout << "Enter the age : ";

std::cin >> this->age;

std::cout << "Enter the height : ";

std::cin >> this->height;

std::cout << "Enter the weight : ";

std::cin >> this->weight;

std::cout << "Enter the sex : ";

std::cin >> this->sex;

}

// Defining the function getdata() to read the values of data members of an object.

void Student ::getdata()

{

std::cout << "The name is : " << this->name << "\n";

std::cout << "The age is : " << this->age << "\n";

std::cout << "The height is : " << this->height << "\n";

std::cout << "The weight is : " << this->weight << "\n";

std::cout << "The sex is : " << this->sex << "\n";

}

int main()

{

// Creating an output stream

std::ofstream OutputFile;

OutputFile.open("09\_object.txt");

Student student;

// Putting data into object

student.putdata();

// Calling the write() function to write an object to a file.

OutputFile.write((char \*)&student, sizeof(student));

std::cout << "\nYour object is successfully stored in the file. \n";

// Closing the output stream

OutputFile.close();

// Creating an input stream

std::ifstream InputFile;

InputFile.open("09\_object.txt");

// Creating an empty object of Student class

Student empty\_student;

std::cout << "\nReading the object from a file : \n";

// Calling the read() function to read an object from a file and transfer its content to an empty object

InputFile.read((char \*)&empty\_student, sizeof(student));

empty\_student.getdata();

InputFile.close();

return 0;

}

**OUTPUT**

PS C:\Users\admin\Documents\Saksham Gupta\MSIT> ./a.exe

Enter the name : Saksham Gupta

Enter the age : 19

Enter the height : 194

Enter the weight : 92

Enter the sex : M

Your object is successfully stored in the file.

Reading the object from a file :

The name is : Saksham Gupta

The age is : 19

The height is : 194

The weight is : 92

The sex is : M

**Program No 10: Write a program to raise an exception if any attempt is made to refer to an element whose index is beyond the array size.**

**THEORY**

**EXCEPTIONS**

An exception is a problem that arises during the execution of a program. A C++ exception is a response to an exceptional circumstance that arises while a program is running, such as an attempt to divide by zero.

Exceptions provide a way to transfer control from one part of a program to another. C++ exception handling is built upon three keywords: **try, catch,** and **throw**.

* **throw** − A program throws an exception when a problem shows up. This is done using a **throw** keyword.
* **catch** − A program catches an exception with an exception handler at the place in a program where you want to handle the problem. The **catch** keyword indicates the catching of an exception.
* **try** − A **try** block identifies a block of code for which particular exceptions will be activated. It's followed by one or more catch blocks.

Assuming a block will raise an exception, a method catches an exception using a combination of the **try** and **catch** keywords. A try/catch block is placed around the code that might generate an exception. Code within a try/catch block is referred to as protected code, and the syntax for using try/catch as follows −

try {

// protected code

} catch( ExceptionName e1 ) {

// catch block

} catch( ExceptionName e2 ) {

// catch block

} catch( ExceptionName eN ) {

// catch block

}

**THROWING EXCEPTIONS**

Exceptions can be thrown anywhere within a code block using **throw** statement. The operand of the throw statement determines a type for the exception and can be any expression and the type of the result of the expression determines the type of exception thrown.

Following is an example of throwing an exception when dividing by zero condition occurs −

double division(int a, int b) {

if( b == 0 ) {

throw "Division by zero condition!";

}

return (a/b);

}

**SOURCE CODE**

#include <iostream> // Header Files

// Sample list against which out of bound exception to be raised

int sample\_list[10] = {1, 2, 3, 5, 7, 11, 13, 17, 19, 23};

int sample\_list\_length = 10;

// Driver Code

int main()

{

int i;

while (true)

{

std::cout << "Enter the index of which element to be accessed: ";

std::cin >> i;

// if entered index out of bounds

if (i < 0 or i >= sample\_list\_length)

throw std::out\_of\_range("Index out of range."); // raise an exception

else

std::cout << "The element at the index is: " << sample\_list[i] << std::endl;

}

}

**OUTPUT**

* Test case 1: Where index is greater than size

PS C:\Users\admin\Documents\Saksham Gupta\MSIT> ./a.exe

Enter the index of which element to be accessed: 1

The element at the index is: 2

Enter the index of which element to be accessed: 4

The element at the index is: 7

Enter the index of which element to be accessed: 6

The element at the index is: 13

Enter the index of which element to be accessed: 8

The element at the index is: 19

Enter the index of which element to be accessed: 10

terminate called after throwing an instance of 'std::out\_of\_range'

what(): Index out of range.

fish: Job 1, './a.out' terminated by signal SIGABRT (Abort)

* Test case 2: Where index is less than 0

PS C:\Users\admin\Documents\Saksham Gupta\MSIT> ./a.exe

Enter the index of which element to be accessed: 2

The element at the index is: 3

Enter the index of which element to be accessed: 4

The element at the index is: 7

Enter the index of which element to be accessed: -3

terminate called after throwing an instance of 'std::out\_of\_range'

what(): Index out of range.

fish: Job 1, './a.out' terminated by signal SIGABRT (Abort)