Scope Resolution Operator (::)

- **Defining member functions outside the class**
  - Tells the compiler **which scope (class, namespace, or global) a function/variable belongs to**.
  - To separate interface (in header) from implementation (in .cpp), we use :: to show which class the function belongs to.

  - // In number.h

```
class Number {
public:
   int factorial(int n);
};
```

Scope Resolution Operator

```
// In number.cpp
#include "number.h"

int Number::factorial(int n) {   // <- scope resolution operator
    int fact = 1;
    for(int i = 1; i <= n; i++) fact *= i;
    return fact;
}
```

- Here, Number::factorial means “This factorial function belongs to the Number class.”

Scope Resolution Operator

- **Accessing global variables when there's a local one.**

```
int x = 10;    // global variable

int main() {
    int x = 5;
    cout << x << endl;    // prints 5 (local x)
    cout << ::x << endl;  // prints 10 (global x, using scope resolution)
}
```

Scope Resolution Operator

- **Namespaces**

```
namespace Math {
   int square(int n) { return n * n; }
}

int main() {
   cout << Math::square(4); // scope resolution to access square inside Math
}
```

- A **namespace** is like a container that groups related functions, classes, and variables together.
- Here, square() belongs to the namespace Math.
- To access it from outside, we use :: (scope resolution).

Access Modifiers

- public elements can be accessed by all other classes and functions.
- private elements cannot be accessed outside the class in which they are declared, except by friend classes and functions.
- protected elements are just like the private, except they can be accessed by derived classes.
- By default, class members in C++ are private, unless specified otherwise.

Specifiers	Same Class	Derived Class	Outside Class
public	Yes	Yes	Yes
private	Yes	No	No
protected	Yes	Yes	No

Constructors

- A constructor is a ‘special’ member function whose task is to initialize the objects of its class. It is special because its name is the same as the class name.
- The constructor is invoked whenever an object of its associated class is created.
- It is called a constructor because it constructs the values of data members of the class.
- In C++, a constructor has the same name as the class, and it does not have a return type.
- Example:

```
class Wall {
        public:
          // create a constructor
          Wall() {
              // code
          }
};
```

- Here, the function Wall() is a constructor of the class Wall. The constructor;
  - has the same name as the class
  - does not have a return type, and
  - is public

Constructors

- A constructor is declared and defined as follows:

```
// class with a constructor
class Integer {
   int m, n;
public:
   Integer(void);   // constructor declared
   ……
   ……
};
Integer :: Integer(void) // constructor definition
{
   m = 0;
   n = 0;
}
```

Constructors: Characteristics

- The constructor functions have some special characteristics. These are:
  - They should be declared in the public section.
  - They are invoked automatically when the objects are created.
  - They do not have return types, not even void, and therefore, they cannot return values.
  - Like other C++ functions, they can have default arguments.

Default Constructor

- A constructor that accepts no parameters is called the default constructor.
- The default constructor for class A is

  A :: A()
- If no such constructor is defined, then the compiler supplies a default constructor.
- Therefore, a statement such as

  A a;

  invokes the default constructor of the compiler to create the object a.

Default Constructor

- A constructor with no parameters is known as a **default constructor**. For example,

```
// C++ program to demonstrate the use of default constructor
#include <iostream>
using namespace std;
// declare a class
class Wall {
   private:
      double length;

   public:
      // default constructor to initialize variable
      Wall()
        : length{5.5} {
        cout << "Creating a wall." << endl;
        cout << "Length = " << length << endl;
      }
};
 int main() {
    Wall wall1;
    return 0;
 }
```

Parameterized Constructors

- The constructor Integer() initializes the data members of all the objects to zero.
- However, in practice, it may be necessary to initialize the various data elements of different objects with different values when they are created.
- C++ permits us to achieve this objective by passing arguments to the constructor function when the objects are created.
- The constructors that can take arguments are called parameterized constructors.

Parameterized Constructors

- The constructor Integer() may be modified to take arguments as shown below:

```
class Integer
{
        int m, n;
            public:
                Integer(int x, int y);        // parameterized constructor
                .....
                .....
};
Integer :: Integer(int x, int y)
{
        m = x; n = y;
}
```

- When a constructor has been parameterized, the object declaration statement, such as
  Integer int1; // may not work.

Parameterized Constructors

- We must pass the initial values as arguments to the constructor function when an object is declared. This can be done in two ways:
  - By calling the constructor explicitly.

    ```
    Integer int1 = Integer(0, 100);      // explicit call
    ```

  - By calling the constructor implicitly.

    ```
    Integer int1(0, 100);          // implicit call
    ```

- Note: When the constructor is parameterized, we must provide appropriate arguments for the constructor.

Parameterized Constructors

- The constructor functions can also be defined as inline functions.

Example:

```
class Integer
{
      int m, n;
     public:
      Integer(int x, int y)          // inline constructor
      {
            m = x; n = y;
      }
      .....
      .....
};
```

- Any function (including constructors) defined inside a class is **inline by default**.
- Writing inline explicitly is redundant.

Parameterized Constructors

- The parameters of a constructor can be of any type except that of the class to which it belongs. The example shown below is illegal.

```
class A
{
      .....
      .....
      public;
      A(A);
};
```

Parameterized Constructors

- However, a constructor can accept a reference to its own class as a parameter. Thus, the statement

```
Class A
{
    .....
    .....
Public:
    A(A&);
};
```

is valid. In such cases, the constructor is called as copy constructor.

Example 1: Parameterized Constructor

```
#include<iostream>
using namespace std;

class Point
{
    int x,y;
    public:
        Point(int a, int b) // inline parameterized constructor definition
        {
            x=a;
            y=b;
        }
    void display()
    {
        cout<<"("<<x<<","<<y<<")\n";
    }
};

int main()
{
    Point p1(1,1);
    Point p2(5,10);
    cout<<"Point p1 =";
    p1.display();
    cout<<"Point p2 =";
    p2.display();
    return 0;
}
```

Output:

```
Point p1 =(1,1)
Point p2 =(5,10)
```

Example 2: Parameterized Constructor

```cpp
// C++ program to calculate the area of a wall

#include <iostream>
using namespace std;

// declare a class
class Wall {
  private:
    double length;
    double height;

  public:
    // parameterized constructor to initialize variables
    Wall(double len, double hgt)
        : length{len}
        , height{hgt} {
    }

    double calculateArea() {
        return length * height;
    }
};
```

C++ Member Initializer List

- Consider the constructor:

```
Wall(double len, double hgt)
  : length{len}
  , height{hgt} {
}
```

- Here,

```
: length{len}
,  height{hgt}
```

is a member initializer list.

- The member initializer list is used to initialize the member variables of a class.

Example 2: Parameterized Constructor

```
int main() {
  // create object and initialize data members
  Wall wall1(10.5, 8.6);
  Wall wall2(8.5, 6.3);

  cout << "Area of Wall 1: " << wall1.calculateArea() << endl;
  cout << "Area of Wall 2: " << wall2.calculateArea();

  return 0;
}
```

Output:

```
Area of Wall 1: 90.3
Area of Wall 2: 53.55
```

C++ Member Initializer List

- The order or initialization of the member variables is according to the order of their declaration in the class rather than their declaration in the member initializer list.
- Since the member variables are declared in the class in the following order:

```
double length;
double height;
```

- The length variable will be initialized first, even if we define our constructor as follows:

```
Wall(double hgt, double len)
  : height{hgt}
  , length{len} {
}
```

Multiple Constructors in a Class

```
Class Integer
{
      int m, n;
   public:
      Integer() {m = 0; n = 0; }    // constructor 1
      Integer(int a, int b)
      {m = a; n = b; }              // constructor 2
      Integer(Integer & i)          // constructor 3
      {m = i.m; n = i.n; }
};
```

- C++ permits us to use more than one constructor in the same class.
- When more than one constructor function is defined in a class, we say that the constructor is overloaded.

friend Function and friend Classes

- Data hiding is a fundamental concept of object-oriented programming. It restricts the access of private members from outside of the class.
- Similarly, protected members can only be accessed by derived classes and are inaccessible from outside. For example,

```
class MyClass {
    private:
        int member1;
}
int main() {
    MyClass obj;
    // Error! Cannot access private members from here.
    obj.member1 = 5;
}
```

- However, there is a feature in C++ called friend functions that breaks this rule and allows us to access member functions from outside the class.
- Similarly, there is a friend class as well.

friend Function

```cpp
// C++ program to demonstrate the working of friend function

#include <iostream>
using namespace std;

class Distance {
    private:
        int meter;

        // friend function
        friend int addFive(Distance);

    public:
        Distance() : meter(0) {}

};

// friend function definition
int addFive(Distance d) {

    //accessing private members from the friend function
    d.meter += 5;
    return d.meter;
}

int main() {
    Distance D;
    cout << "Distance: " << addFive(D);
    return 0;
}
```

Output:

```
Distance: 5
```

friend Function

- A friend function can access the private and protected data of a class. We declare a friend function using the friend keyword inside the body of the class.

```
class className {
   … .. …
   friend returnType functionName(arguments);
   … .. …
}
```

- A friend function is most useful when we need to operate on objects of two different classes.

friend Class

- We can also use a friend Class in C++ using the friend keyword. For example,

```
class ClassB;

class ClassA {
  // ClassB is a friend class of ClassA
  friend class ClassB;
  ... .. ...
}

class ClassB {
  ... .. ...
}
```

- When a class is declared as a friend class, all the member functions of the friend class become friend functions.
- Since ClassB is a friend class, we can access all members of ClassA from inside ClassB.
- However, we cannot access members of ClassB from inside ClassA. It is because friend relation in C++ is only granted, not taken.

friend Class

```cpp
// C++ program to demonstrate the working of friend class

#include <iostream>
using namespace std;

// forward declaration
class ClassB;

class ClassA {
    private:
        int numA;

        // friend class declaration
        friend class ClassB;

    public:
        // constructor to initialize numA to 12
        ClassA() : numA(12) {}
};
```

friend Class

```
class ClassB {
    private:
        int numB;

    public:
        // constructor to initialize numB to 1
        ClassB() : numB(1) {}

    // member function to add numA
    // from ClassA and numB from ClassB
    int add() {
        ClassA objectA;
        return objectA.numA + numB;
    }
};

int main() {
    ClassB objectB;
    cout << "Sum: " << objectB.add();
    return 0;
}
```

Output:

```
Sum: 13
```

Overloaded Constructors

Complex.h

```
#ifndef COMPLEX_H
#define COMPLEX_H

class Complex {
    float x, y;

public:
    Complex();                      // Constructor with no arguments
    Complex(float a);               // Constructor with one argument
    Complex(float real, float imag); // Constructor with two arguments

    // Friend functions
    friend Complex sum(Complex, Complex);
    friend void show(Complex);
};

#endif
```

- The first time the preprocessor sees #ifndef COMPLEX_H, it finds that COMPLEX_H is not defined → it defines it and includes the file.
- The second time, since COMPLEX_H is already defined, the header is **skipped**.
- So the compiler sees the class only once.

Overloaded Constructors

Complex.cpp

```cpp
#include <iostream>
#include "complex.h"
using namespace std;

// Constructor with no arguments
Complex::Complex() { }

// Constructor with one argument
Complex::Complex(float a) {
    x = y = a;
}

// Constructor with two arguments
Complex::Complex(float real, float imag) {
    x = real;
    y = imag;
}

// Friend function to sum two complex numbers
Complex sum(Complex c1, Complex c2) {
    Complex c3;
    c3.x = c1.x + c2.x;
    c3.y = c1.y + c2.y;
    return c3;
}

// Friend function to display a complex number
void show(Complex c) {
    cout << c.x << " + j" << c.y << "\n";
}
```

Overloaded Constructors

Main.cpp

```cpp
#include <iostream>
#include "complex.h"
using namespace std;

int main() {
    Complex A(2.7, 3.5);  // define and initialize
    Complex B(1.6);       // define and initialize
    Complex C;            // define
    C = sum(A, B);        // sum() is a friend

    cout << "A = "; show(A); // show() is also a friend
    cout << "B = "; show(B);
    cout << "C = "; show(C);

    // Another way to give initial values
    Complex P, Q, R;      // define P, Q, and R
    P = Complex(2.5, 3.9); // initialize P
    Q = Complex(1.6, 2.5); // initialize Q
    R = sum(P, Q);

    cout << "\n";
    cout << "P = "; show(P);
    cout << "Q = "; show(Q);
    cout << "R = "; show(R);

    return 0;
}
```

Output:

```
A = 2.7 + j3.5
B = 1.6 + j1.6
C = 4.3 + j5.1

P =2.5 + j3.9
Q =1.6 + j2.5
R =4.1 + j6.4
```

Constructors with Default Arguments

- It is important to distinguish between the default constructor A::A() and the default argument constructor A::A(int=0).
- The default argument constructor can be called with either one argument or no arguments. When called with no arguments, it becomes a default constructor.
- When both these forms are used together in a class, it causes ambiguity for a statement such as A a;
- The ambiguity is whether to ‘call’ A::A() or A::A(int=0). Therefore, two such constructors should not be defined in a class at the same time.

Dynamic Initialization of Objects

- Objects can be initialized dynamically, too. That is, the initial value of an object may be provided during run time.
- One advantage of dynamic initialization is that we can provide various initialization formats, using overloaded constructors.
- Consider the long-term deposit schemes working in the commercial banks. The banks provide different interest rates for different schemes as well as for different periods of investment.

Dynamic Initialization of Objects

Fixed_deposit.h

```
#ifndef FIXED_DEPOSIT_H
#define FIXED_DEPOSIT_H

class Fixed_Deposit {
    long int p_amount;    // Principal amount
    int years;            // Period of investment
    float rate;           // Interest rate
    float r_value;        // Return value of amount

public:
    Fixed_Deposit();                                    // Default constructor
    Fixed_Deposit(long int p, int y, float r = 0.12);   // Decimal rate constructor
    Fixed_Deposit(long int p, int y, int r);            // Percent rate constructor

    void display(void);
};

#endif
```

- The program illustrates how to use the class variables for holding account details and how to construct these variables at run time using dynamic initialization.

Dynamic Initialization of Objects

Fixed_deposit.cpp

```cpp
#include <iostream>
#include "fixed_deposit.h"
using namespace std;

// Default constructor
Fixed_Deposit::Fixed_Deposit() { }

// Constructor with decimal rate
Fixed_Deposit::Fixed_Deposit(long int p, int y, float r) {
    p_amount = p;
    years = y;
    rate = r;
    r_value = p_amount;
    for (int i = 1; i <= y; i++) {
        r_value = r_value * (1.0 + r);
    }
}

// Constructor with percent rate
Fixed_Deposit::Fixed_Deposit(long int p, int y, int r) {
    p_amount = p;
    years = y;
    rate = r;
    r_value = p_amount;
    for (int i = 1; i <= y; i++) {
        r_value = r_value * (1.0 + float(r) / 100);
    }
}

// Display function
void Fixed_Deposit::display(void) {
    cout << "\n"
         << "Principal Amount = " << p_amount << "\n"
         << "Return value     = " << r_value << "\n";
}
```

Dynamic Initialization of Objects

Main.cpp

```cpp
#include <iostream>
#include "fixed_deposit.h"
using namespace std;

int main() {
    Fixed_Deposit FD1, FD2, FD3;    // deposits created
    long int p;                     // principal amount
    int y;                          // investment period (years)
    float r_d;                      // interest rate in decimal form
    int r_p;                        // interest rate in percent form

    cout << "Enter amount, period, interest rate (in percent):\n";
    cin >> p >> y >> r_p;
    FD1 = Fixed_Deposit(p, y, r_p);

    cout << "Enter amount, period, interest rate (decimal form):\n";
    cin >> p >> y >> r_d;
    FD2 = Fixed_Deposit(p, y, r_d);

    cout << "Enter amount and period:\n";
    cin >> p >> y;
    FD3 = Fixed_Deposit(p, y);

    cout << "\nDeposit 1";
    FD1.display();

    cout << "\nDeposit 2";
    FD2.display();

    cout << "\nDeposit 3";
    FD3.display();

    return 0;
}
```

Dynamic Initialization of Objects

Output:

```
Enter amount, period, interest rate (in percent)
10000 3 18
Enter amount, period, interest rate (decimal form)
10000 3 0.18
Enter amount and period
10000 3

 Deposit1
Principal Amount = 10000
Return value = 16430.3

 Deposit2
Principal Amount = 10000
Return value = 16430.3

 Deposit3
Principal Amount = 10000
Return value = 14049.3
```

Copy Constructor

- The copy constructor in C++ is used to copy data from one object to another.

Wall.h

```
#ifndef WALL_H
#define WALL_H

class Wall {
private:
    double length;
    double height;

public:
    Wall(double len, double hgt);     // Parameterized constructor
    Wall(const Wall& obj);            // Copy constructor

    double calculateArea();           // Function to calculate area
};

#endif
```

Copy Constructor

- The copy constructor in C++ is used to copy data from one object to another.

Wall.cpp

```cpp
#include "wall.h"

// Parameterized constructor
Wall::Wall(double len, double hgt)
    : length{len}, height{hgt} {
}

// Copy constructor
Wall::Wall(const Wall& obj)
    : length{obj.length}, height{obj.height} {
}

// Function to calculate area
double Wall::calculateArea() {
    return length * height;
}
```

Main.cpp

```cpp
#include <iostream>
#include "wall.h"
using namespace std;

int main() {
    // create an object of Wall class
    Wall wall1(10.5, 8.6);

    // copy contents of wall1 to wall2 using copy constructor
    Wall wall2 = wall1;

    // print areas of wall1 and wall2
    cout << "Area of Wall 1: " << wall1.calculateArea() << endl;
    cout << "Area of Wall 2: " << wall2.calculateArea() << endl;

    return 0;
}
```

Output:

```
Area of Wall 1: 90.3
Area of Wall 2: 90.3
```

Copy Constructor

- We then assign the values of the variables of the obj object to the corresponding variables of the object, calling the copy constructor. This is how the contents of the object are copied.
- In main(), we then create two objects wall1 and wall2 and then copy the contents of wall1 to wall2:
- // copy contents of wall1 to wall2

  Wall wall2 = wall1;
- Here, the wall2 object calls its copy constructor by passing the reference of the wall1 object as its argument.

Copy Constructor

Code.h

```
#ifndef CODE_H
#define CODE_H

class Code {
    int id;

public:
    Code();                 // Default constructor
    Code(int a);            // Parameterized constructor
    Code(Code &x);          // Copy constructor

    void display(void);     // Display function
};

#endif
```

code.cpp

```
#include <iostream>
#include "code.h"
using namespace std;

// Default constructor
Code::Code() { }

// Parameterized constructor
Code::Code(int a) {
    id = a;
}

// Copy constructor
Code::Code(Code &x) {
    id = x.id;
}

// Display function
void Code::display(void) {
    cout << id;
}
```

Copy Constructor

Main.cpp

```cpp
#include <iostream>
#include "code.h"
using namespace std;

int main() {
    Code A(100);      // Object A is created and initialized
    Code B(A);        // Copy constructor called
    Code C = A;       // Copy constructor called again

    Code D;           // D is created, not initialized
    D = A;            // Assignment operator (not copy constructor)

    cout << "\n id of A: "; A.display();
    cout << "\n id of B: "; B.display();
    cout << "\n id of C: "; C.display();
    cout << "\n id of D: "; D.display();

    return 0;
}
```

Output:

```
id of A: 100
id of B: 100
id of C: 100
id of D: 100
```

- A reference variable has been used as an argument to the copy constructor. We cannot pass the argument by value to a copy constructor.

Dynamic Constructors

- In cases like strings, arrays, or matrices, different objects may need different amounts of memory.
- The constructors can also be called to allocate memory while creating objects.
- This will enable the system to allocate the right amount of memory for each object when the objects are not of the same size, thus resulting in the saving of memory.
- Allocation of memory to objects at the time of their construction is known as dynamic construction of objects.
- The memory is allocated with the help of the new operator.

Dynamic Constructors

string_class.h

```
#ifndef STRING_CLASS_H
#define STRING_CLASS_H

class String {
    char *name;
    int length;

public:
    String();                   // Default constructor
    String(const char *s);      // Parameterized constructor

    void display(void);         // Display function
    void join(String &a, String &b); // Join two strings
};

#endif
```

Dynamic Constructors

string_class.cpp

```cpp
#include <iostream>
#include <cstring>
#include "string_class.h"
using namespace std;

// Default constructor
String::String() {
    length = 0;
    name = new char[length + 1];
    name[0] = '\0';  // Initialize as empty string
}

// Parameterized constructor
String::String(const char *s) {
    length = strlen(s);
    name = new char[length + 1];
    strcpy(name, s);
}

// Display function
void String::display(void) {
    cout << name << "\n";
}

// Join function
void String::join(String &a, String &b) {
    length = a.length + b.length;
    delete[] name;  // Correct way to free dynamically allocated array
    name = new char[length + 1];
    strcpy(name, a.name);
    strcat(name, b.name);
}
```

Dynamic Constructors

main.cpp

```
#include <iostream>
#include "string_class.h"
using namespace std;

int main() {
    const char *first = "Joseph ";
    String name1(first), name2("Louis "), name3("Lagrange"), s1, s2;

    s1.join(name1, name2);
    s2.join(s1, name3);

    name1.display();
    name2.display();
    name3.display();
    s1.display();
    s2.display();

    return 0;
}
```

Output:

```
Joseph
Louis
Lagrange
Joseph Louis
Joseph Louis Lagrange
```

Constructing Two-Dimensional Arrays

matrix.h

```
#ifndef MATRIX_H
#define MATRIX_H

class Matrix {
    int **p;   // Pointer to Matrix
    int d1, d2;

public:
    Matrix(int x, int y);                  // Constructor
    void get_element(int i, int j, int value);
    int& put_element(int i, int j);    // Return reference
    ~Matrix();                             // Destructor (to free memory)
};

#endif
```

- int **p; means p will point to an array of pointers, and each pointer will point to an array of integers → structure of a matrix in dynamic memory.

Constructing Two-Dimensional Arrays

matrix.cpp

```cpp
#include <iostream>
#include "matrix.h"
using namespace std;

// Constructor
Matrix::Matrix(int x, int y) {
    d1 = x;
    d2 = y;
    p = new int*[d1];    // allocate an array of int* (row pointers)
    for (int i = 0; i < d1; i++) {
        p[i] = new int[d2];    // for each row pointer, allocate
    }                          // an array of int (columns)
}

// Set element
void Matrix::get_element(int i, int j, int value) {
    p[i][j] = value;//Stores a value into the matrix at position (i, j).
}

// Get element (by reference)
int& Matrix::put_element(int i, int j) {
    return p[i][j]; //Returns the reference to the element at (i, j).
}

// Destructor
Matrix::~Matrix() {
    for (int i = 0; i < d1; i++) {
        delete[] p[i];  // free each row
    }
    delete[] p;  // free row pointers
}
```

- First, new int*[d1] creates an array of d1 pointers (each pointer will represent one row).
- Then, inside the loop, each p[i] is assigned memory for d2 integers.
- A **d1 × d2 matrix** is dynamically allocated in heap memory.

Constructing Two-Dimensional Arrays

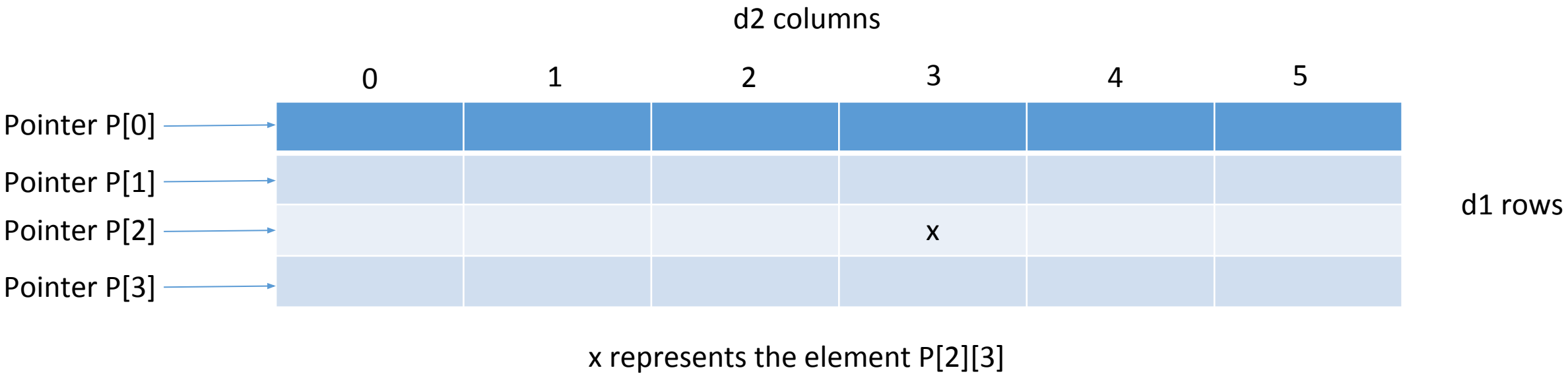
main.cpp

```cpp
#include <iostream>
#include "matrix.h"
using namespace std;

int main() {
    int m, n;
    cout << "Enter size of matrix: ";
    cin >> m >> n;

    Matrix A(m, n);

    cout << "Enter matrix elements row by row:\n";
    int value;

    for (int i = 0; i < m; i++) {
        for (int j = 0; j < n; j++) {
            cin >> value;
            A.get_element(i, j, value);
        }
    }

    cout << "\nElement at (1,2): " << A.put_element(1, 2) << endl;

    return 0;
}
```

Output:

```
Enter size of matrix:3 4
Enter matrix elements row by row
11 12 13 14
15 16 17 18
19 20 21 22

17
```

Constructing Two-Dimensional Arrays



x represents the element P[2][3]

- The constructor first creates a vector pointer to an **int** of size **d1**. Then, it allocates, iteratively an **int** type vector of size d2 pointed at by each element p[i]. Thus, space for the elements of a d1xd2 matrix is allocated from free store as shown above.

What is Stack Memory?

- **Automatically managed memory**: allocated when a function is called, freed when it returns.
- **Stores:**
  - Local variables (int x, y;)
  - Function parameters (main(int argc, char* argv[]))
  - Return addresses (where to go back after a function call)
  - Objects created without new
- **Fast allocation and deallocation** (just moves the stack pointer).
- **Limited size** (a few MBs, depending on OS).
- **Scope-bound**: memory is released automatically when a variable goes out of scope.
- No need for delete — compiler manages it.

What is Heap Memory?

- **Manually managed memory**: allocated with new, freed with delete.
- **Stores:**
  - Objects and arrays you explicitly allocate at runtime.
  - Used when you need memory that persists beyond a function call.
- Controlled by a programmer (or smart pointers in modern C++).
- **Flexible size** (much larger than stack, limited by system RAM).
- Slower allocation than stack (because the OS has to find free space).
- **Must be freed explicitly** using delete or delete[].
- Memory leaks if you forget to release it.

Const OBJECTS

- We can create X as a constant object of the class matrix as follows:

  const matrix X(m, n); // object X is constant

- Any attempt to modify the values of **m** and **n** will generate a compile-time error. Further, a constant object can call only const member functions.
- Whenever **const** objects try to invoke **non-const** member functions, the compiler generates errors.

Move Constructor

- The move constructor (introduced with the C++11 standard) is a recent addition to the family of constructors in C++.
- It is like a copy constructor that constructs the object from the already existing objects, but instead of copying the object in the new memory, it makes use of move semantics to transfer the ownership of the already created object to the new object without creating extra copies.
- It can be seen as stealing the resources from other objects.
- The move constructor uses std::move() to transfer ownership of resources and it is called when a temporary object is passed or returned by value.
- Syntax:

```
className (className&& obj) {
    // body of the constructor
}
```

Move Constructor

- MyClass constructor takes an **rvalue reference (int &&a)** and moves the value of **a** into the **private member b** using **std::move.**
- The main function creates an integer a, and then moves it into obj1 by calling the move constructor.
- The move constructor takes the rvalue reference of the object of the same class and transfers the ownership of this object to the newly created object.
- Like a copy constructor, the compiler will create a move constructor for each class that does not have any explicit move constructor.

```cpp
#include <iostream>
#include <vector>
using namespace std;

class MyClass {
private:
    int b;

public:
    // Constructor
    MyClass(int &&a) : b(move(a)) {
        cout << "Move constructor called!" << endl;
    }

    void display() {
            cout << b <<endl;
    }
};

int main() {
    int a = 4;
    MyClass obj1(move(a));  // Move constructor is called

    obj1.display();
    return 0;
}
```

Output:

```
Move constructor called!
4
```

Destructors

- A destructor, as the name implies, is used to destroy the objects that have been created by a constructor.
- Like a constructor, the destructor is a member function whose name is the same as the class name but is preceded by a tilde.
- A destructor never takes any arguments, nor does it return any value.
- It will be invoked implicitly by the compiler upon exit from the program (or block or function, as the case may be) to clean up storage that is no longer accessible.
- It is a good practice to declare destructors in a program since it releases the memory space for future use.

Destructors

- Whenever a **new** is used to allocate memory in the constructors, we should use **delete** to free that memory.
- For example, the destructor for the **Matrix** class may be defined as follows:

```
Matrix:: ~Matrix()
{
    for (int i=0; i<d1;i++)
    delete p[i];
    delete p;
}
```

- This is required because when the pointers to objects go out of scope, a destructor is not called implicitly.

Implementation of Destructors

```
#include<iostream>
using namespace std;

int count=0;

class Testing
{
    public:
        Testing()
        {
            count++;
            cout<<"\n\nConstructor Msg: Object number "<<count<< " created..";
        }
        ~Testing()
        {
            cout<<"\n\nDestructor Msg: Object number "<<count<< " destroyed..";
            count--;
        }
};
```

Implementation of Destructors

```
int main()
{
    cout<<"Inside the main block...";
    cout<<"\n\nCreating first object T1 ...";

    Testing T1;
    {
        //Block B1
        cout<<"\n Inside Block 1..";
        cout<<"\n \n Creating two more objects T2 and T3..";
        Testing T2, T3;
        cout<<"\n \n Leaving Block 1..";
    }
    cout<<"\n\n Back inside the main block..";
    return 0;
}
```

Output:

```
Inside the main block...

Creating first object T1 ...

Constructor Msg: Object number 1 created..
 Inside Block 1..

 Creating two more objects T2 and T3..

Constructor Msg: Object number 2 created..

Constructor Msg: Object number 3 created..

 Leaving Block 1..

Destructor Msg: Object number 3 destroyed..

Destructor Msg: Object number 2 destroyed..

 Back inside the main block..

Destructor Msg: Object number 1 destroyed..
```

Implementation of Destructors

Note

- A Class constructor is called every time an object is created. Similarly, as the program control leaves the current block, the objects in the block start getting destroyed, and the destructors are called for each one of them.
- The objects are destroyed in the reverse order of their creation.
- Finally, when the main block is exited, destructors are called corresponding to the remaining objects present inside the main.

Implementation of Destructors

```
#include<iostream>
using namespace std;

class Testing
{
    int *a;
    public:
    Testing(int size)
    {
        a = new int[size];
        cout<<"\n\n Constructor Msg: Integer array of size: "<< size <<"created..";
    }
    ~Testing()
    {
        cout<<"\n\n Destructor Msg: Freed up the memory allocated for integer array ";
    }
};

int main()
{
    int s;
    cout<<"\n Enter the size of the array:";
    cin>>s;
    cout<<"\n\n Creating an object of test class..";
    Testing T(s);
    cout<<"\n\n Press any key to end the Program..";
    return 0;
}
```

Implementation of Destructors

Output:

```
Enter the size of the array:5


Creating an object of test class..

Constructor Msg: Integer array of size: 5created..

Press any key to end the Program..

Destructor Msg: Freed up the memory allocated for integer array
```