Basic Concepts: Constructors and Destructors

What is a Constructor?

- **Constructor** is a special member function automatically called when an object is created.
- It has the same name as the class and **no return type**.
- Used to initialize objects.

Types:

- **Default constructor**: No arguments.
- **Parameterized constructor**: Accepts parameters.
- **Copy constructor**: Initializes an object from another object of the same class.

What is a Destructor?

- **Destructor** is a special member function that is called automatically when an object goes out of scope or is deleted.
- Used to release resources.

A **friend function** and **friend class** in C++ are powerful features that allow specific external functions or entire classes to access the private and protected data members of another class. These features are useful for certain scenarios where two or more classes or functions need to collaborate closely, bypassing the usual encapsulation rules of C++.

Friend Function

- A **friend function** is a function that is **not a member** of a class but is allowed access to that class's private and protected members.
- It is declared inside the class using the `friend` keyword.
- Friend functions can be regular (global) functions or a member function of another class.
- The declaration is made inside the class; the definition is outside

Friend Class

- A **friend class** is a class whose all member functions have access to the private and protected members of another class.
- Declared using `friend class ClassName;` inside the class where access should be granted

1. Default, Parameterized, and Copy Constructors

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

class Box {
   int length;
public:
   // Default constructor
   Box() {
      length = 0;
      cout << "Default constructor called\n";
   }
   // Parameterized constructor
   Box(int l) {
      length = l;
      cout << "Parameterized constructor called\n";
   }
   // Copy constructor
   Box(const Box &b) {
      length = b.length;
      cout << "Copy constructor called\n";
   }
   void show() {
      cout << "Length: " << length << endl;
   }
};

int main() {
   Box b1;        // Default constructor
   Box b2(10);    // Parameterized constructor
   Box b3 = b2;   // Copy constructor
   b1.show();
   b2.show();
   b3.show();
   return 0;
}
```

2. Constructor Overloading

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

class Point {
   int x, y;
public:
   // Default constructor
   Point() : x(0), y(0) {}
   // Parameterized constructor (int, int)
   Point(int a, int b) : x(a), y(b) {}
   // Parameterized constructor (single int)
   Point(int a) : x(a), y(0) {}

   void show() {
      cout << "(" << x << ", " << y << ")" << endl;
   }
};

int main() {
   Point p1;         // Default
   Point p2(5, 7);    // Two ints
   Point p3(8);       // Single int

   p1.show();
   p2.show();
   p3.show();
   return 0;
}
```

3. Constant Objects with Constructors

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

class Circle {
   const double radius;
public:
   // Constructor initializing constant data member
   Circle(double r) : radius(r) {}
   double area() const {
      return 3.14 * radius * radius;
   }
};
int main() {
   const Circle c1(5.0); // Constant object
   cout << "Area: " << c1.area() << endl;
   // c1.radius = 20; // Error: cannot modify a const object
   return 0;
}
```

4. Dynamic Constructor (with Memory Allocation)

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

class Array {
   int *arr;
   int size;
public:
   // Dynamic constructor
   Array(int s) {
      size = s;
      arr = new int[size];
      cout << "Dynamic memory allocated\n";
      for(int i = 0; i < size; ++i) arr[i] = i;
   }
   void show() {
      for(int i = 0; i < size; ++i) cout << arr[i] << " ";
      cout << endl;
   }
   // Destructor to free memory
   ~Array() {
      delete[] arr;
      cout << "Memory freed\n";
   }
};
int main() {
   Array a1(5);
   a1.show();
   // Destructor will be called automatically
   return 0;
}
```

5. Role of Destructors

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

class Demo {
public:
   Demo() {
      cout << "Constructor called\n";
   }
   ~Demo() {
      cout << "Destructor called\n";
   }
};

int main() {
   Demo d;
   cout << "Inside main function\n";
   return 0;
}
// When main ends, destructor called automatically
```

6. Operator Overloading (+, ==) Using Member Functions

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

class Complex {
   int real, imag;
public:
   Complex(int r = 0, int i = 0) : real(r), imag(i) {}

   // Overloading +
   Complex operator+(const Complex &c) {
      return Complex(real + c.real, imag + c.imag);
   }
   // Overloading ==
   bool operator==(const Complex &c) {
      return (real == c.real && imag == c.imag);
   }
   void display() {
      cout << real << " + " << imag << "i" << endl;
   }
};

int main() {
   Complex c1(2, 3), c2(2, 3), c3;
   c3 = c1 + c2;
   c3.display();
   if (c1 == c2)
      cout << "Complex numbers are equal" << endl;
```

```
    else
        cout << "Complex numbers are not equal" << endl;
    return 0;
}
```

7. Use of this Pointer to Resolve Naming Conflicts

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

class Sample {
    int num;
public:
    void setNum(int num) {
        this->num = num; // 'this' distinguishes member from parameter
    }
    void show() {
        cout << "Number is: " << num << endl;
    }
};

int main() {
    Sample s;
    s.setNum(10);
    s.show();
    return 0;
}
```

8. Friend Function and Friend Class

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

class Box;

class Display {
public:
    void showVolume(Box &);
};

class Box {
    int length;
public:
    Box(int l) : length(l) {}
    friend void Display::showVolume(Box &); // Friend class function
    friend void show(Box &);                // Friend function
};

void Display::showVolume(Box &b) {
    cout << "Volume (assuming cube): " << b.length * b.length * b.length << endl;
}
```

```
void show(Box &b) { // Friend function
    cout << "Length: " << b.length << endl;
}

int main() {
    Box b(3);
    Display d;
    d.showVolume(b);
    show(b);
    return 0;
}
```