fit@hcmus

# Object-Oriented Programming

## Object Life Cycle


Tran Duy Hoang

# Topics Covered

- **Constructors**
- Deconstructor
- Static Members
- Class Template

# Constructors

- Used to **initialize** objects
- Called when an object is **created**
- Has the **same name** as the class
- Has **no return** type
- Can be **overloaded**

# Example: Constructors

```
class Fraction
{
private:
    int num;
    int denom;
public:
    Fraction();
    Fraction(int num, int denom);
    Fraction(int value);
    Fraction(string str);
    Fraction(const Fraction &frac);
};
```

```
int main()
{
    Fraction frac1;
    Fraction frac2(1, 2);
    Fraction frac3(3);
    Fraction frac4("1/2");
    Fraction frac5(frac1);
    Fraction* frac6 = new Fraction();
    Fraction* frac7 = new Fraction(1, 2);
    Fraction* frac8 = new Fraction(3);
    Fraction* frac9 = new Fraction("1/2");
    Fraction* frac10 = new Fraction(frac1);
}
```

# Types of Constructors

- Default Constructor
  - A constructor with no parameters
  - E.g., Fraction frac();
- Parameterized Constructor
  - A constructor that takes arguments
  - E.g., Fraction frac(1, 2);
- Copy Constructor
  - A constructor that copies the values of an existing object
  - E.g., Fraction frac2(frac1);

# Default Constructor

- A constructor that takes **no arguments**
- Used to initialize an object with **default values**
- If **no constructor** is defined

  → C++ provides an implicit default constructor

# Example: Default Constructor

```
class Fraction
{
private:
    int num;
    int denom;
public:
    Fraction();
};

Fraction::Fraction()
{
    this->num = 0;
    this->denom = 1;
}
```

```
int main()
{
    Fraction frac1;
    Fraction* frac2 = new Fraction();
}
```

# Parameterized Constructor

- A constructor that takes **arguments**
- Allows initialize an object with **specific values**
- Can be **overloaded**
  → multiple constructors having different parameters

# Example: Parameterized Constructor

```
class Fraction
{
private:
    int num;
    int denom;
public:
    Fraction(int num, int denom);
    Fraction(int value);
};

Fraction::Fraction(int num, int denom)
{
    this->num = num;
    this->denom = denom;
}
```

```
Fraction::Fraction(int value)
{
    this->num = value;
    this->denom = 1;
}

int main()
{
    Fraction frac1(1, 2);
    Fraction frac2(3);
    Fraction* frac3 = new Fraction(1, 2);
    Fraction* frac4 = new Fraction(3);
}
```

# Copy Constructor

- Initializes a new object by **copying** an existing object
- Takes **a reference to an object** of the same class
- Called when:
  - A new object is initialized from an existing object
  - An object is passed by value to a function
  - An object is returned by value from a function
- If **no copy constructor** is defined

  → C++ provides an implicit copy constructor

# Example: Copy Constructor

```
class Fraction
{
private:
    int num;
    int denom;
public:
    Fraction(const Fraction &frac);
};

Fraction::Fraction(const Fraction &frac)
{
    this->num = frac.num;
    this->denom = frac.denom;
}
```

```
int main()
{
    Fraction frac1;
    Fraction frac2(frac1);
    Fraction* frac3 = new Fraction(frac2);
}
```

# Advices: Constructors

- A class should have at least 3 constructors
  - Default constructor
  - Copy constructor
  - Parameterized constructor initializes all attributes

```
class Fraction
{
private:
    int num;
    int denom;

public:
    Fraction();
    Fraction(const Fraction &frac);
    Fraction(int num, int denom);
};
```

# Topics Covered

- Constructors
- **Deconstructor**
- Static Members
- Class Template

# Destructor

- Used to **clean up** resources
  - like dynamically allocated memory
- Called when an object
  - **goes out** of scope or
  - is explicitly **deleted**
- Has the **same name** as the class with a **tilde (~)** prefix
- Take **no arguments** and has **no return** type
- Each class can have **only one** destructor
  - it cannot be overloaded

# Example: Destructor

```
class Array
{
private:
    int length;
    int* data;
public:
    Array(int length);
    ~Array();
};

Array::Array(int length)
{
    this-> length = length;
    this->data = new int[length];
}
```

```
Array::~Array()
{
    delete[] this->data;
}

int main()
{
    Array arr;
    Array *pArr = new Array();
    delete pArr;
}
```

# Topics Covered

- Constructors
- Deconstructor
- **Static Members**
- Class Template

# Static Members

- Belong to **class** rather than any **object**
- Shared across all objects of the class
- Examples of applications
  - Generate unique IDs for each object
  - Count how many objects have been created
  - Manage application-wide (global) settings
  - Manage logs in a centralized way

# Static Members

- Use keyword "*static*"
- Does not have a "*this*" pointer
- Called using either the class name or an object
  - use the scope resolution operator (*::*)
- Static attributes
  - shared among all objects of the class
  - has only one copy in memory
  - must be initialized outside the class
- Static function
  - can only access static attributes

# Example: Static Members

```
class Fraction {
private:
    static int count;
public:
    static void printCount();
};

Fraction::Fraction() {
    count++; // Fraction::count++;
}

Fraction::~Fraction() {
    count--; // Fraction::count--;
}
```

```
void Fraction::printCount() {
    cout << "Total objects: " << count;
}

int Fraction::count = 0; // initialization

int main() {
    Fraction frac1;
    Fraction* frac2 = new Fraction();
    frac1.printCount;
    delete frac2;
    Fraction::printCount();
}
```

# Topics Covered

- Constructors
- Deconstructor
- Static Members
- **Class Template**

# Class Template

- Allow a class work with **any data type**
- Helps in
  - code reusability
  - type safety
  - code duplication reduction

# Example: Static Class

```
template <class T>
class Array
{
private:
    int size;
    T* data;
public:
    Array(int size);
    T& getElement(int i);
    T findMax();
    ~Array();
};
```

```
int main()
{
    Array<int> arr1(10);
    int a = arr1.getElement(5);
    int max1 = arr1.FindMax();

    Array<Fraction> arr2(10);
    Fraction b = arr2.getElement(5);
    Fraction max2 = arr2.FindMax();
}
```

# Exercise 3.1

- Provide class Fraction with constructors to
  - initialize a fraction = 0 by default
  - initialize a fraction with num and denom
  - initialize a fraction with an integer value
  - initialize a fraction with an string (“num/denom”)
    - E.g., Fraction frac(“1/2”)
  - initialize a fraction from another fraction

# Exercise 3.2

- Based on Exercise 3.1, extend class Fraction with the ability to count the number of fraction objects that are created in the main() function.

# Exercise 3.3

- Provide class Array with constructors and destructor to
  - initialize an array with zero length by default
  - initialize an array with length and all elements = 0
  - initialize an array with int[] and length
    - E.g., int a[] = {1,2,3}; Array arr(a, 3);
  - initialize an array with from another object of Array
    - E.g., Array arr1; Array arr2(arr1);
  - Depose an array without memory leak

# Exercise 3.4

- Based on Exercise 3.3, extend class Array to include elements of any data type.