Object Oriented Programming with C++

7. Constructors and Destructors

By: Prof. Pandav Patel

Second Semester, 2020-21
Computer Engineering Department
Dharmsinh Desai University

Constructors

```
#include<iostream>

class Complex
{
    float re;
    float im;
public:
    void add(Complex c);
    void print();
};
void Complex::add(Complex c)
{
    re += c.re;
    im += c.im;
}
void Complex::print()
{
    std::cout << re << " + j";
    std::cout << im << std::endl;
}
```

```
int main()
{
    Complex c1;
    c1.print();
    return 0;
}
```

```
-1.25547e+33 + j4.59163e-41
```

Constructors

- How to initialize objects?
  - In C++ objects can be initialized using constructors
  - Constructor is a special member function of a class that is used to initialize objects of its class type
    - Name of the constructor is same as the class name
    - It does not have any return type (not even void)
  - Constructor should be declared in public section (though there are use cases for defining constructor in private section, we will not look into those)
  - Constructor can be defined within class definition or outside
  - Constructor is invoked whenever object of its associated class is created
  - More than one constructors could be defined for a class (overloading)
  - Constructors can have default arguments

Constructors

- Default Constructor
  - When no constructor has been defined for a class, compiler supplies a default constructor
    - Default constructor provided by compiler is “do-nothing” kind
    - As soon as a constructor is defined for a class (with or w/o args), compiler no longer provides default constructor

Constructors

```
#include<iostream>

class Complex
{
    float re;
    float im;
public:
    // Constructor declaration
    Complex(float x, float y);
    void add(Complex c);
    void print();
};
// Constructor definition
Complex::Complex(float x, float y)
{
    re = x;
    im = y;
}
```

```
void Complex::add(Complex c)
{
    re += c.re;
    im += c.im;
}
void Complex::print()
{
    std::cout << re << " + j";
    std::cout << im << std::endl;
}

int main()
{
    Complex c1(1.1, 1.1);
    Complex c2 = Complex(2.2, 2.2);
    // error: no matching function for call to ‘Complex::Complex()
    Complex c3;
    c3.add(c1);
    c3.add(c2);
    c3.print();
    return 0;
}
```

Constructors

```
#include<iostream>
class Complex
{
    float re;
    float im;
public:
    // Default Constructor
    Complex() {
        re = 0;
        im = 0;
    }
    // Constructor declaration
    Complex(float x, float y); //Overload
    void add(Complex c);
    void print();
};
// Constructor definition
Complex::Complex(float x, float y)
{
    re = x;
    im = y;
}
```

```
void Complex::add(Complex c)
{
    re += c.re;
    im += c.im;
}
void Complex::print()
{
    std::cout << re << " + j";
    std::cout << im << std::endl;
}

int main()
{
    Complex c1(1.1, 1.1);
    Complex c2 = Complex(2.2, 2.2);
    Complex c3;
    c3.add(c1);
    c3.add(c2);
    c3.print();
    return 0;
}
```

In copy initialization below
- A temporatry object is created
- Temporary object is copied into new object being created
- Temporary object is destroyed

But modern compilers will directly create new object and skip the temporary object (copy elision). Unless code is compiled with **-fno-elide-constructors** flag

// Direct Initialization
// Copy Initialization
// Default constructor

```
3.3 + j3.3
```

Constructors

```
#include<iostream>
class Complex
{
    float re;
    float im;
public:
    // Constructor declaration
    // Default constructor
    // Default arguments
    Complex(float x = 0, float y = 0);
    void add(Complex c);
    void print();
};
// Constructor definition
Complex::Complex(float x, float y)
{
    re = x;
    im = y;
}
void Complex::add(Complex c)
{
    re += c.re;
    im += c.im;
}
void Complex::print()
{
    std::cout << re << " + j";
    std::cout << im << std::endl;
}

int main()
{
    Complex c1(1.1, 1.1);
    Complex c2 = Complex(2.2, 2.2);
    Complex c3;
    c3.add(c1);
    c3.add(c2);
    c3.print();
    return 0;
}
```

```
3.3 + j3.3
```

Constructors

```
#include<iostream>
class Complex
{
    float re;
    float im;
public:
    // Constructor declaration
    // Default constructor
    Complex(float x = 0, float y = 0);
    void add(Complex c);
    void print();
};
// Constructor definition
Complex::Complex(float x, float y)
{
    re = x;
    im = y;
}
void Complex::add(Complex c)
{
    re += c.re;
    im += c.im;
}
void Complex::print()
{
    std::cout << re << " + j";
    std::cout << im << std::endl;
}
int main()
{
    Complex c1(1.1, 1.1);
    // Memory for objects can be allocated dynamically too
    // and constructor will be invoked for it too
    Complex *c2 = new Complex(2.2, 2.2);
    Complex &c3 = *(new Complex);
    c3.add(c1);
    c3.add(*c2);
    c3.print();
    return 0;
}
```

```
3.3 + j3.3
```

```
#include<iostream>
using std::cout;
using std::endl;
class Test {
    int i;
    float j;
};
```

```
int main() {
    // Common mistake
    // Test t0(); // This is considered as function declaration
    Test t0;
    Test t1 = Test();
    Test *t2 = new Test();
    //Test t3(2, 3);
    //Test t4 = Test(2, 3);
    //Test *t5 = new Test(2, 3);
    //Test t6(2);
    //Test t7 = Test(2);
    //Test *t8 = new Test(2);
    Test t9{};
    Test t10 = Test{};
    Test *t11 = new Test{};
    //Test t12{2, 3};
    //Test t13 = Test{2, 3};
    //Test *t14 = new Test{2, 3};
    //Test t15{2};
    //Test t16 = Test{2};
    //Test *t17 = new Test{2};
    //Test t18 = {2, 3};
    //Test t19 = {2};
    Test t20 = {};
    return 0;
}
```

t3-t8 and t12-t19 can not be constructed with the given syntax, as members i and j are private and there is no constructor which can initialize them

```
#include<iostream>
using std::cout;
using std::endl;
class Test {
public:
    int i;
    float j;
};
```

```
int main() {
    // Common mistake
    // Test t0(); // This is considered as function declaration
    Test t0;
    Test t1 = Test();
    Test *t2 = new Test();
    //Test t3(2, 3);
    //Test t4 = Test(2, 3);
    //Test *t5 = new Test(2, 3);
    //Test t6(2);
    //Test t7 = Test(2);
    //Test *t8 = new Test(2);
    Test t9{};
    Test t10 = Test{};
    Test *t11 = new Test{};
    Test t12{2, 3};
    Test t13 = Test{2, 3};
    Test *t14 = new Test{2, 3};
    Test t15{2};
    Test t16 = Test{2};
    Test *t17 = new Test{2};
    Test t18 = {2, 3};
    Test t19 = {2};
    Test t20 = {};
    return 0;
}
```

t12-t19 can now be constructed with the given syntax, as members i and j are public now, and you can think of this as old C style initialization. But t3-t8 can not be constructed as that syntax requires constructor

```
#include<iostream>
using std::cout;
using std::endl;
class Test {
    int i;
    float j;
public:
    Test(int x = 0, int y =0) {
        cout << "Constructor called\n";
        i = x;
        j = y;
    }

};
```

When constructor is present then all initializations will call constructor, instead of old C-style initialization.

```
int main() {
    // Common mistake
    // Test t0(); // This is considered as function declaration
    Test t0;
    Test t1 = Test();
    Test *t2 = new Test();
    Test t3(2, 3);
    Test t4 = Test(2, 3);
    Test *t5 = new Test(2, 3);
    Test t6(2);
    Test t7 = Test(2);
    Test *t8 = new Test(2);
    Test t9{};
    Test t10 = Test{};
    Test *t11 = new Test{};
    Test t12{2, 3};
    Test t13 = Test{2, 3};
    Test *t14 = new Test{2, 3};
    Test t15{2};
    Test t16 = Test{2};
    Test *t17 = new Test{2};
    Test t18 = {2, 3};
    Test t19 = {2};
    Test t20 = {};
    return 0;
}
```

```
Constructor called
Constructor called
Constructor called
Constructor called
Constructor called
Constructor called
Constructor called
Constructor called
Constructor called
Constructor called
Constructor called
Constructor called
Constructor called
Constructor called
Constructor called
Constructor called
Constructor called
Constructor called
Constructor called
Constructor called
Constructor called
```

Constructors

- Copy Constructor
  - A copy constructor of class **T** has first parameter as **T&**, **const T&**, **volatile T&**, or **const volatile T&**, and either there are no other parameters, or the rest of the parameters all have default values
    - All of our copy constructors will have **T&** or **const T&** as first parameter
    - Use **const T&**, unless you know that you need **T&**
  - When no copy constructor has been defined for a class, compiler supplies a default copy constructor
    - Default copy constructor provided by compiler does byte-wise copy from one object to another object
  - For copy constructor of class **T**, we are passing reference variable of type **T** as first arg. We can not pass argument by value, because it will lead to infinite loop

Constructors

- Copy Constructor
  - The copy constructor is called whenever an object is initialized from another object of the same type, which includes
    - initialization: ***T a = b;*** or ***T a(b);***, where ***b*** is of type ***T***
    - function argument passing: ***f(a);***, where ***a*** is of type ***T*** and **f** is ***return-type f(T t);***
    - function return: ***return a;*** inside a function such as ***T f()***, where ***a*** is of type ***T***

    - ***T a = T();***
      - ***T()*** creates *temporary object*
      - *Temporary object* is copied into object ***a*** using **copy constructor**
      - *Temporary object* is destroyed
      - But modern compilers will directly create new object and skip the temporary object (copy elision). Unless code is compiled with **-fno-elide-constructors** flag

Constructors

```
#include<iostream>

class Complex
{
    float re;
    float im;
public:
    Complex(float x = 0, float y = 0);
    // Temporary object is const
    // So, error while initializing c3 and c5,
    // if first argument of copy constructor is not const
    Complex(const Complex &c)
    {
        std::cout << "Copy " << std::endl;
        re = c.re;
        im = c.im;
    }
};
Complex::Complex(float x, float y)
{
    std::cout << "Constructor " << std::endl;
    re = x;
    im = y;
}
```

```
Complex c1; // Default initialization
int main()
{
    std::cout << "main function 1\n";
    Complex c2(2, 2); // Direct initialization
    std::cout << "main function 2\n";
    // With flag -fno-elide-constructors
        // copy temporary object to c3
    // Without flag -fno-elide-constructors
        // Direct initialization of c3
    Complex c3 = Complex(3, 3);
    std::cout << "main function 3\n";
    Complex c4(c1); // Copy initialization
    std::cout << "main function 4\n";
    // With flag -fno-elide-constructors
        // Copy constructor will be called twice
        // to create temporary object and to copy to c5
    // Without flag -fno-elide-constructors
        // Directly copy to c5
    Complex c5 = Complex(c1);
    std::cout << "main function 5\n";
    Complex c6 = {6, 6}; // Direct initialization of C6
    std::cout << "main function 6\n";
    Complex c7 = {c1}; // Copy initialization
    std::cout << "main function 7\n";
    return 0;
}
```

Without flag

```
Constructor
main function 1
Constructor
main function 2
Constructor
main function 3
Copy
main function 4
Copy
main function 5
Constructor
main function 6
Copy
main function 7
```

With flag

```
Constructor
main function 1
Constructor
main function 2
Constructor
Copy
main function 3
Copy
main function 4
Copy
Copy
main function 5
Constructor
main function 6
Copy
main function 7
```

Constructors

```
#include<iostream>
class Complex{
    float re;
    float im;
public:
    Complex(float x = 0, float y = 0);
    Complex(const Complex &c){
        std::cout << "Copy " << std::endl;
        re = c.re;
        im = c.im;
    }
    Complex add(Complex c);
    void print();
};
Complex::Complex(float x, float y){
    std::cout << "Constructor " << std::endl;
    re = x;
    im = y;
}
Complex Complex::add(Complex c){
    re += c.re;
    im += c.im;
    return *this;
}
void Complex::print(){
    std::cout << re << " + j";
    std::cout << im << std::endl;
}
```

```
Complex c1; // Default initialization

int main()
{
    std::cout << "main function 1\n";
    Complex c2 = c1; // copy initialization
    std::cout << "main function 2\n";
    c2 = c1; // assignment, because object already exists
    std::cout << "main function 3\n";
    // With flag -fno-elide-constructors
        // copy arg, copy return object to temp object
        // copy temp object to c3
    // Without flag -fno-elide-constructors
        // copy arg, copy return object to c3
    Complex c3 = c1.add(c2);
    std::cout << "main function 4\n";
    // copy arg, copy return object to temp object
    // assign temp object to c3
    c3 = c1.add(c2);
    std::cout << "main function 5\n";
    return 0;
}
```

Without flag

```
Constructor
main function 1
Copy
main function 2
main function 3
Copy
Copy
main function 4
Copy
Copy
main function 5
```

With flag

```
Constructor
main function 1
Copy
main function 2
main function 3
Copy
Copy
Copy
main function 4
Copy
Copy
main function 5
```

3.3 + j3.3

Constructors

```
#include<iostream>
class Test
{
    const int t;
    int i;
public:
    Test(int num)
    {
        // error: assignment of read-only member ‘Test::t’
        t = num;
        i = num;
    }
    void print();
};
```

```
void Test::print()
{
        std::cout << "t = " << t << ", i = " << i << std::endl ;
}
int main()
{
    Test t1(10);
    t1.print();
    return 0;
}
```

- **const** members can not be initialized inside the constructor body
  - **const** members must be initialized before entering constructor body
  - Because object already exists (with all its members already created) when constructor body starts executing
  - Object is destroyed only after destructor body finishes its execution
    - Hence **this** keyword can be used in constructor and destructor bodies

Constructors

```
#include<iostream>
class Test
{
    const int t;
    int i;
public:
    Test(int num): t(num), i(num * 2)
    {
        i = num * 3;
    }
    void print();
};
```

```
void Test::print()
{
        std::cout << "t = " << t << ", i = " << i << std::endl;
}
int main()
{
    Test t1(10);
    t1.print();
    return 0;
}
```

```
t = 10, i = 30
```

- In C++, constructors can have initializer list
  - Initializer list is provided during constructor definition
    - Colon sybmol is provided after constructor prototype, and then initializer list is provided, and then constructor body starts
    - Initializer list is comma separated. Each entry in the list initializes one member. Name of the member is followed by parentheses, and expression is present inside parantheses. Expression will be evaluated and final value will be used to initialize the corresponding member
  - Initializer list can be used to initialize **const** members of the object
  - Initializer list is acted upon before object comes into existence
  - Initializer list has other advantages as well (e.g. can initialize reference variables)
  - Initializer list can be used to initialize non-const members too
  - It is not mandatory to initialize all members in initializer list

```
#include<iostream>
using std::cout;
using std::endl;
class Test {
    int i;
public:
    Test(int i) {
        this->i = i;
    }
    void print() {
        cout << i << endl;
    }
};
int main() {
    Test t0;
    t0.print();
    Test t1(3);
    t1.print();
    return 0;
}
```

Q1:
Will this code compile fine? If no, why?
If yes, what will be the output?

```
#include<iostream>
using std::cout;
using std::endl;
class Test {
    int i;
public:
    Test(int i = 0) {
        this->i = i;
    }
    void print() {
        cout << i << endl;
    }
};
int main() {
    Test t0;
    t0.print();
    Test t1(3);
    t1.print();
    return 0;
}
```

Q2:
Will this code compile fine? If no, why?
If yes, what will be the output?

```
#include<iostream>
using std::cout;
using std::endl;
class Test {
    int i;
public:
    Test(int i) {
        this->i = i;
    }
    void print() {
        cout << i << endl;
    }
};
int main() {
    Test t0(3);
    t0.print();
    Test t1(t0);
    t1.print();
    return 0;
}
```

Q3:
Will this code compile fine? If no, why?
If yes, what will be the output?

Will compiler provide default copy constructor in this case?

```
#include<iostream>
using std::cout;
using std::endl;
class Test {
    int i;
public:
    Test(const Test &t) {
        this->i = t.i;
    }
    void print() {
        cout << i << endl;
    }
};
int main() {
    Test t0;
    t0.print();
    Test t1(t0);
    t1.print();
    return 0;
}
```

Q4:
Will this code compile fine? If no, why?
If yes, what will be the output?

Will compiler provide default constructor in this case?

```
#include<iostream>
using std::cout;
using std::endl;
class Test {
    int i;
public:
    Test(const Test &t) {
        this->i = t.i * 2;
    }
    Test(int i = 0) {
        this->i = i;
    }
    void print() {
        cout << i << endl;
    }
};
int main() {
    Test t0;
    t0.print();
    Test t1 = Test(3);
    t1.print();
    return 0;
}
```

Q5:
Will this code compile fine? If no, why?
If yes, what will be the output?

Result will be different if compiled with
-fno-elide-constructors flag

```
#include<iostream>
using std::cout;
using std::endl;
class Test {
    const int t;
    int i;
public:
    Test(): i(3) {
    }
    void print() {
        cout << i << " " << t << endl;
    }
};
int main() {
    Test t0;
    t0.print();
    return 0;
}
```

Q6:
Will this code compile fine? If no, why?
If yes, what will be the output?

Can we leave const member uninitialized?

Can we initialize const member within constructor body?

```
#include<iostream>
using std::cout;
using std::endl;
class Test {
    const int t;
    int i;
public:
    Test(): i(3), t(i * 2) {
    }
    void print() {
        cout << i << " " << t << endl;
    }
};
int main() {
    Test t0;
    t0.print();
    return 0;
}
```

Q7:
Will this code compile fine? If no, why?
If yes, what will be the output?

Can other members be used in **expression within brackets** in initializer list? For example using member **i** to initialize member **t**.

Which member will be initialized first? **i** or **t**?

What will be the values of **i** and **t**?

```
#include<iostream>
using std::cout;
using std::endl;
class Test {
    const int t;
    int i;
public:
    Test(): i(t * 2), t(3) {
    }
    void print() {
        cout << i << " " << t << endl;
    }
};
int main() {
    Test t0;
    t0.print();
    return 0;
}
```

Q8:
Will this code compile fine? If no, why?
If yes, what will be the output?

```
#include<iostream>
using std::cout;
using std::endl;
class Test {
    const int t;
    int i;
public:
    Test(int num): i(t * num), t(3) {
    }
    void print() {
        cout << i << " " << t << endl;
    }
};
int main() {
    Test t0(5);
    t0.print();
    return 0;
}
```

Q9:
Will this code compile fine? If no, why?
If yes, what will be the output?

Can we mix parameters and other class members in expression within brackets in initializer list? For example **num** and **t**.

```
#include<iostream>
using std::cout;
using std::endl;
class Test {
    int i;
    int &ref;
public:
    Test(): ref(i), i(3) {
        i = 50;
    }
    void print() {
        cout << i << " " << ref << endl;
    }
};
int main() {
    Test t0;
    t0.print();
    return 0;
}
```

Q10:
Will this code compile fine? If no, why?
If yes, what will be the output?

Can reference be initialized in initializer list?

Can we initialize reference within constructor body and not in intializer list? This is a question for you.

Identify memory leak in this code

```
#include<iostream>
#include<cstring>
using std::cout;
using std::endl;
class Str {
    int len;
    char *ptr;
public:
    Str(const char *ptr) {
        len = strlen(ptr);
        this->ptr = new char[len + 1];
        strcpy(this->ptr, ptr);
    }
    void print() {
        cout << ptr << endl;
    }
};
int main() {
    for(int i = 0; i < 5; i++) {
        Str st1(" Hello world, this is just an example!");
        cout << i;
        st1.print();
    }
    return 0;
}
```

```
#include<iostream>
#include<cstring>
using std::cout;
using std::endl;
class Str {
    int len;
    char *ptr;
public:
    Str(const char *ptr) {
        cout << "Constructor called" << endl;
        len = strlen(ptr);
        this->ptr = new char[len + 1];
        strcpy(this->ptr, ptr);
    }
    void print() {
        cout << ptr << endl;
    }
    ~Str() {
        cout << "Destructor called" << endl;
        delete ptr;
    }
};
int main() {
    for(int i = 0; i < 5; i++)
    {
        Str st1(" Hello world, this is just an example!");
        cout << i;
        st1.print();
    }
    return 0;
}
```

```
Constructor called
0 Hello world, this is just an example!
Destructor called
Constructor called
1 Hello world, this is just an example!
Destructor called
Constructor called
2 Hello world, this is just an example!
Destructor called
Constructor called
3 Hello world, this is just an example!
Destructor called
Constructor called
4 Hello world, this is just an example!
Destructor called
```

Destructors

- A destructor is a special member function that is called just before the lifetime of an object ends.
  - Object’s lifetime ends only after its destructor finishes its execution. Execution of destructor is the last thing in the lifetime of an object.
  - The destructor is called whenever an object's lifetime ends, which includes
    - program termination, for objects with static storage duration (static and global)
    - end of scope, for objects with automatic storage duration
    - delete-expression, for objects with dynamic storage duration

- The purpose of the destructor is to free the resources (e.g. Memory, files etc.) that the object may have acquired during its lifetime.
  - For example, if memory is allocated dynamically for object members, it must be released before lifetime of object ends. Destructor is the obvious choice to do it.

Destructors

- If no user-declared destructor is provided for a class then compiler will provide default destructor

- Name of the destructor is same as the name of the class, but is preceded by tilde
  - ~class_name();
- Like constructor, destructor does not return any value, hence no return statement in destructor body
- Unlike constructor, destructor does not have any parameters, hence it can not be overloaded and there can only be one destructor per class
- Destructor can be defined within class definition or outside class definition
  - If it is defined outside class definition then like other member functions, name of the class and scope resolution operator is used for full qualification

- Auto and global objects are destroyed in reverse order than order of their creation

Destructors

```
#include<iostream>
class Test{
        int *ip;
public:
        Test(int i = 0) {
                ip = new int;
                *ip = i;
                std::cout << "Constructor called ";
                std::cout << *ip << std::endl;
        }
        void print() {
                std::cout << *ip << std::endl;
        }
        ~Test() {
                std::cout << "Destructor called ";
                std::cout << *ip << std::endl;
                delete ip;
        }
};
```

```
Test gt;

int main() {

        std::cout << "Inside main function\n";

        Test lt(9);
        Test &dt = *(new Test(5));

        gt.print();
        lt.print();
        dt.print();

        delete &dt;
        std::cout << "main is about to end\n";
        return 0;
}
```

```
Constructor called 0
Inside main function
Constructor called 9
Constructor called 5
0
9
5
Destructor called 5
main is about to end
Destructor called 9
Destructor called 0
```

const object and const method

```
#include<iostream>
#include<cstring>
using std::cout;
using std::endl;
class Str
{
    int len;
    char *strng;
public:
    void print()
    {
        cout << len << ": ";
        puts(strng);
    }
    Str(const char *strng)
    {
        len = strlen(strng);
        this->strng = new char[len + 1];
        strcpy(this->strng, strng);
        cout << "constructor: " << strng << endl;
    }
    ~Str()
    {
        cout << "Destructor: " << strng << endl;
        delete strng;
    }
};
```

```
void fun(Str &ln)
{
    ln.print();
    cout << "Before deleting dynamic object\n";
    delete &ln;
    cout << "After deleting dynamic object\n";
}

int main()
{
    Str first_name("Jalaj");
    const Str middle_name("Pandavkumar");
    Str *last_name = new Str("Patel");
    first_name.print();
    // error: passing ‘const Test’ as ‘this’
    // argument discards qualifiers
    // const obj can't call non-const method
    middle_name.print();
    fun(*last_name);
    last_name->print();
    return 0;
}
```

- Data members (private or public) of const object can not be changed. Neither by member function nor from outside the object
- const objects can call only const member functions
- Const member functions have const keyword after prototype (during definition)
- const member functions can not change any data member of the object (even when object is non-const)
- Const methods can have auto variables and they can change
- const member functions can be invoked by non-const objects
- When method is not changing any data members of the object, it is good practice to declare it const

Destructors

```
#include<iostream>
#include<cstring>
using std::cout;
using std::endl;
class Str
{
    int len;
    char *strng;
public:
    void print() const
    {
        cout << len << ": ";
        puts(strng);
    }
    Str(const char *strng)
    {
        len = strlen(strng);
        this->strng = new char[len + 1];
        strcpy(this->strng, strng);
        cout << "constructor: " << strng << endl;
    }
    ~Str()
    {
        cout << "Destructor: " << strng << endl;
        delete strng;
    }
};
```

```
void fun(Str &ln)
{
    ln.print();
    cout << "Before deleting dynamic object\n";
    delete &ln;
    cout << "After deleting dynamic object\n";
}

int main()
{
    Str first_name("Jalaj");
    const Str middle_name("Pandavkumar");
    Str *last_name = new Str("Patel");
    first_name.print();



    middle_name.print();
    fun(*last_name);
    last_name->print(); //Undefined behavior
    return 0;
}
```

```
constructor: Jalaj
constructor: Pandavkumar
constructor: Patel
5: Jalaj
11: Pandavkumar
5: Patel
Before deleting dynamic object
Destructor: Patel
After deleting dynamic object
1029472992:
Destructor: Pandavkumar
Destructor: Jalaj
```

Destructors

- If we do not release dynamic memory allocated for object’s data members, then it will lead to memory leaks
  - Once object’s lifetime ends, it’s destructor will be called.
  - After execution of destructor, object’s memory would be freed.
  - If dynamic memory allocated for data members has not been freed already and if we do not free it in destructor, then it can not be freed until end of program execution.
    - Because we will lose pointer to dynamic memory allocated for data member, as pointer is stored in data member of object.
- If object itself has been created dynamically, on calling delete for that object, destructor would be called for it
  - Object can be created dynamically (with new keyword) in one function and can be deleted (with delete keyword) in another function.
  - Lifetime of dynamic object is in developer’s control.
  - Once dynamic object is deleted, accessing it is undefined behaviour and it must be avoided.

Interesting topics

- These topics are not covered in class, but they are very interesting
  - Initializer list – data members are initialized in order in which they are declared in class and not in order in which they are present in the initializer list (could be understood with Rectangle and Point classes)
  - Object within another object – When a class X contains object of class Y as its member, constructor of Y will be called before constructor of X. And destructor of Y will be called before destructor of X. (could be understood with Rectangle and Point classes)
  - shallow copy Vs deep copy

Interesting reads

- Uniform Initialization in C++11
  - https://www.geeksforgeeks.org/uniform-initialization-in-c/
  - https://stackoverflow.com/questions/24953658/what-are-the-differences-between-c-like-constructor-and-uniform-initialization
- Copy initialization Vs Direct Initialization
  - https://stackoverflow.com/questions/1051379/is-there-a-difference-between-copy-initialization-and-direct-initialization
- Copy elision in C++
  - https://www.geeksforgeeks.org/copy-elision-in-c/
- When do we use Initializer List in C++?
  - https://www.geeksforgeeks.org/when-do-we-use-initializer-list-in-c/
- Copy constructor
  - https://en.cppreference.com/w/cpp/language/copy_constructor