Functions: operator new() and operator delete()

- We can also write operator functions for new and delete
- C++ also every operator has its corresponding operator fn version.
- C++ also allows operator new and operator delete fn to dynamically allocate and de-allocate mem.

ex: `int *p = (int *) malloc(sizeof(int));`

```
*p = 5
free(p);
```

```
int *p = (int *) operator new(sizeof(int));
*p = 5;
operator delete(p);
```

* For user defined type they way operator new version works and corresponding operator fn works differs major.

Dynamically managed ~~memory~~ in array in C++

```
int *a = (int *) malloc(sizeof(int)*a);
free(a);
```

(77) using operator new[]

```
int *a = new int[3]
```

↑ How many elements to allocate

```
delete[] a;
```

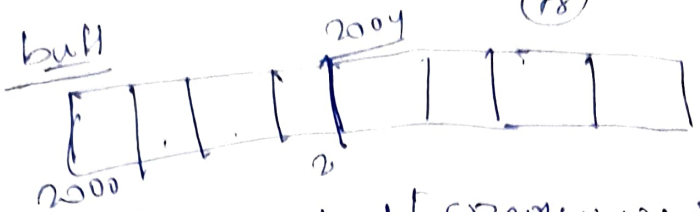
Note: Operator new[] and Operator new are different

Operator new()

Placement new in C++:

memory manager
free storage
from where or to which location we are getting the memory or not in our control

- By placement new we can allocate memory from known storage like.
- Dynamically allocate from stack or global area.
- placement new operator takes a buffer and to place objects
- These are not dynamically allocated on Heap maybe allocated on Heap
- Allocation by placement new must not be deleted.



unsigned char buff[sizeof(int)*2]; // Buffer as stack.

```
int *p2int = new(buff) int(3);
```

allocates 4 byte starting from buff

```
int *q2int = new (buff+sizeof(int)) int(5);
```

without using placement news

```
int *pBuff = (int*) (buff+0);
int *qBuff = (int*) (buff+sizeof(int));
```

* For placement new we have to provide the buffer.

mixing malloc and operator new

allocator	De-allocator
malloc()	free()
operator new	operator delete
operator new[]	operator delete[]
operator new()	No delete.

(10) passing NULL pointer to delete operator is secure.

Overloading Operator new

(I) The first parameter of overloaded Operator new must be size_t

(II) return type of overloaded new must be void *

```
void * operator new(size_t, n)   // there can be more parameter also
{
    void *ptr;
    ptr = malloc(n)
    return ptr;
}
```

since we have passed int type. so compiler will automatic convert to

```
void operator delete(void *p)
{
    free(p);
}
```

void * to int * while return

```
int main()
{
    int *p = new int;
    *p = 30;
    delete p;
}
```

from int type compiler define the size and pass it

calling overloaded operator new

// calling overloaded operator delete

Overloading operator delete

- first parameter to overloaded operator delete must be void*
- operator delete should not be overloaded (usually) with extra parameters

Overloading operator new[]

```
void * operator new[](size_t n, char setv) // initialize allocated array with setv
{
    void *ptr = operator new(n);
    memset(ptr, setv, n);
    return ptr;
}

void operator delete[](void *s)
{
    operator delete(s);
}
```

```
char *t = new('#') char[10];
```

allocate array of 10 elements and fill with #

```
delete[] t;
```

(51) Classes and objects

C code

typedef struct complex
{ double re, im;
} complex;

complex n1 = {4.2, 5.3}

class Complex
{ public:
double re, im;
};

Creating object

Complex c1 = {4.2, 5.3} is not

sometimes

Complex c1(4.2, 5.3)

public members are

It requires Constructor

it works as long as

Complex n1 = {4.2, 5.3}; or Complex c1(

or

Complex n1(4.2, 5.3);

It like struct initialization

This pointer

- an implicit this pointer holds the address of an object
- this pointer serves as identity of the object in C++
- Type of this pointer for class X
  Object is: X * const this
- accessible only in methods

class X
{ public: ~~int int~~;
int m1, m2;
void f(int k1, int k2)
{ m1 = k1; // implicit access w/o this pointer
this->m1 = k2 → explicit

cout << this << endl;

}

int main()

{ X a;

a.f(2,3);

~~Address~~ cout << &a << endl;

}

In source code

```
class X { void f(int,
    int); ... }

X a; a.f(2,3);
```

In Binary code

```
void X::f(X * const this,
    int, int);

X::f(&a, 2, 3) //
                &a = this
```

Use of this

- to distinguish member from non member

Return the Object

```
Complex & inc()
{
    ++re;
    ++im;
    return *this;
}
```

* In C++ Objects do not have any separate identity.

Structure of an Object

- is determined by the combined value of all its data member.

Complex c = { 4.3, 5.3 } →

c = { 4.3, 5.9 }

Access specifiers in C++

ex.1

```
class complex
{   public:
        double re, im;

    public:
        double norm()
        {   return sqrt(re*re + im*im);
        }
};

void print(const Complex &t)
{   cout << t.re << t.im << endl;
}

int main()
{   Complex C = {4.2, 5.3}; //Okay
    print()
    cout << C.norm();
}
```

global fn

since re is public

private case

```
class complex
{   private:
        double re, im;

    public:
        double() norm()
        {   return sqrt(re*re
                + im*im)
        }
};
```

```
void print() // global fn

{
    void print ( const Complex &t)
    {
        cout << t.re << t.im << endl;
    }
```

// cannot access private members outside of class

```
int main()
{
    Complex c{4.2, 3.5}; // Error
```

since second is more private mem.

```
    print()
}
```

Important point

* class provide access specifier for members (data as well as functions) to enforce data hiding that separates implementation from interface.

* **private:** accessible inside defn of class i.e inside member fn of some class

* **public:** accessible everywhere
  (a) member fn of same class
  (b) " " " " data "
  (c) global fn.

(85)
By default access specifier of class in C++ is private.

Information Hiding / Encapsulation

(i) private part of the class (attributes or methods) form its implementation because the class alone should be concerned with it and have the right to change it

(ii) public part of the class (attributes or methods) constitutes its interface which is available to all other for using the class

(iii) Customarily we put all attributes in private part and methods in public part this ensures

(a) The state of object can only be changed through one of its methods (can with the knowledge of the class)

(b) behaviour of an object is available to others through the methods

This is known as information hiding.

Design guideline.

protecting the state behind the private visibility and exposing the behaviour through public visibility is known as info hiding / encapsulation or state based design.

Design

(I) All declaration in Header file

(II) Implementation of class in .cpp and Application in separate file.

File: stack.h

```
class Stack
{
    private:
        char *data_;
        int top;
    public:
        Stack();
        ~Stack();
        void push(char x)
        void pop()
        char top()
};
#include "stack.h"
```

File stack.cpp: Implementation

```
Stack :: Stack() : data_(new char[100]),
{                  top_(-1) { }

Stack :: ~Stack() { delete [] data_; }

int Stack :: empty()
{ return (top_ == -1); }
```

```
#include <stdio.h> //Apps.cpp
int main()
{

}
```

Will the app see private data? ie

Application cannot access private data. Will the application get to see what private member does the class have.

Get-set methods - idioms for fine grained access control.

Why straightaway we are not making data as public but providing public method to manipulate the data?

If we make data public straight away instead of providing public method then Object will never get to know when this data member is read/written. But if it is done through method then if it is read it can or also do some computations after/read/written. So later.

(88) the object will always be alive here data member is changed

DOB, password

Constructor

- is a member fn with this pointer.
- Name same as class Name

```
class stack
{   public:
        stack();
};
```

- Has no return type

```
stack :: stack();
```

- No return : Here has no return stmt

```
stack() :: stack() : top_(-1)
{ } // Returns implicit
```

- Initializer list is used to initialize data member

```
stack :: stack() :
        data_(new char[10]),
        top_(-1)
{ }
```

Initializer list

↑ Array contains stmt

also

(5) Implicit call by instantiation / operator new

Stack s

may have any no of parameter

can be overloaded

Ex:-

```
Stack
{   private:
        char *data_,
        int top;
    Stack(); // constructor
};
Stack::Stack() : data_(new char[10]),
                 top_(-1) {
    cout << "Stack:: Stack() called" <<
        endl;
}
int main()
{   Stack s; // (constructor called
    char str[10] = "ABCDE"
    for (int i=0; i<=5; i++)
        s.push(str[i])
    while(!s.empty())
    {   cout<<s.top()<<endl;
        s.pop();
    }
}
```

Parameterized constructor

```
class Complex
{
    private:
        double re, im;

    public:
        Complex(double re, double im):
        re(re), im(im) {}
};

int main()
{
    Complex c(4.2, 5.35);

    Complex d = {2.15, 9.13};
}
```

Brace Same →

* If data mem is private then brace same provided

Constructor with default parameter

```
class Complex
{
    private:
        double re, im;

    public:
        Complex(double re = 0.0,
        double im = 0.0): re(re),
        im(im) {}
};
```

(91) Complex c(4.2, 3.15), d(4.2), e;

or

```
class Complex
{   private:
        double re, im;
    public:
        Complex(double re=0.0,
                double im=0.0);
};
Complex::Complex(double re, double
    im) : re(re), im(im){};
```

Constructor Overloading

```
class Complex
{   private:
        double re, im;
    public:
        Complex(double re, double im)
        : re(re), im(im){}
        Complex(double re) : re(re),
            im(0.0){}
        Complex() : re(0.0), im(0.0)
            {}
};
```

Complex c(4.25, 3.15), c2(3.12), c3;

Destructor

- It member fn with same name as class preceded by ~
- class Stack
  {
  public:
  ~Stack()
  };
- Has no return type
  Stack::~Stack();
- implicitly called at end of the scope by operator delete. May be explicitly called by the object (rare)
  {
  Stack s
  } // calls Stack::~Stack(&s);
- No parameter allowed - unique for the class
- Cannot be overloaded

Ex:
```
class Stack
{
    private:
        char *stk;
        int top;
    public:
    ~Stack()
    {
        cout << "Destructor" << endl;
        delete stk;
    }
};
```

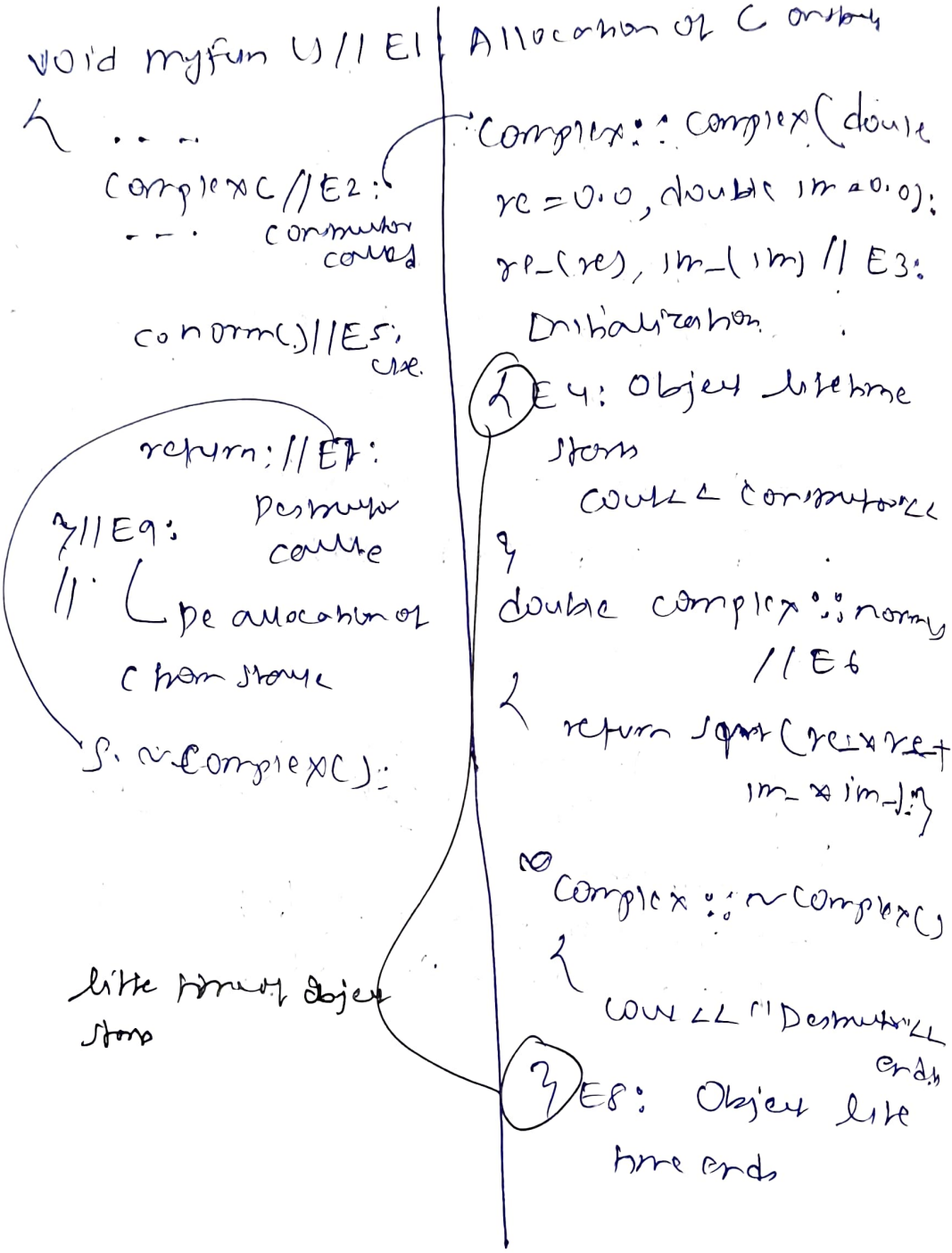
(93) Default constructor / Destructor

- A constructor with no parameter called default constructor
- If no constructor is provided by user compiler supplies a free default constructor
- compiler provided default constructor can not initialize the Object with proper value. It has no code in its body.
- Default constructor (free or user defined) are required to define array of Objects

Default destructor

- If no destructor is provided by the user the compiler supplies a free default destructor.
- compiler provided (default) destructor has no code in its body.

When is an object ready and how long can it be used?



* Before the body of constructor start i.e just after the completion of initialization list Object lifetime starts i.e it is ready for use.

* Object lifetime does not start when constructor is called but it starts when the constructor has completed its initialization list part.

(95) * from E2 to E3 Object may have inconsistent value.

* E3 is right before the return but at the return destructor is called.

* from the beginning of body of constructor and at the end of body of destructor Object is alive.

why do we need initialization ? as we can initialize the data member as part of body of constructor?

There are 2 of parameters to Object and in purpose of initialization list we can initialize in whatever any order. if initialization of one data member depends on other then it matters to have their order of initialization. If we write in the body of constructor then we have to maintain the particular order.

If we initialize data members in initialization list then the compiler follows unique approach the compiler initialize the data member in the order in which data members are written in the class not in order we write them in initialization list

ex. Complex::Complex(double re, double im) : im_(im), re_(re){ // Here also re_ get Cout<<"constructor"<<endl; initialized first and then im_ initialized since re_

(96) debug better im- in class Complex like

```
class Complex
{
    double re_, im_;
};
```

Object Lifetime

Execution stage:

① memory allocation and Binding

↓ stack, global common area statically

Heap

Association bern name of the object and the address that we are using for that object in the memory. For all diff type of Object lifetime happen in different ways

(ii) constructor call and Execution

(iii) Object use:

(iv) Destructor call and execution

(v) memory de-allocation and De-binding

Object lifetime.

① starts with execution of constructor body

as soon as initialization ends and control enters constructor body

must follow memory allocation

(22) * Ends with execution of Destructor body
- as soon as control leave destructor body
- must precede new de-allocate.

* For Objects of a built in type / predefined
- No explicit constructor/destructor
- Lifetime spans from object definition to end of scope

constructor/destructor follow LIFO order.

* destructor is called in the reverse order of constructor

Object lifetime for automatic

```
class complex
{
    private:
        double re_, im_;
    public:
    complex( double re=0.0, double
    im=0.0): re_(re), im_(im) //
    {
        cout << "Constructor" << "re=" << re_ << "
        im=" << im_ << endl;
    }
    ~complex()
    {
        cout << " destructor" << "re="
        << re_ << " im=" << im_ << endl;
    }
    double norm()
    {
        return sqrt(re_*re_ + im_*
        im_);}
```

```
void print()
{
    cout<<"|"<<re<<"+j"<<im<<"|";
    endl;
}
};
int main()
{
    complex c(4.2, 5.3), d(2.4);
    c.print();
    d.print();
    return 0;
}
```

O/p:

Constructor re = 4.2 im = 5.3

Constructor re = 2.4 im = 0.0

|4.2+j5.3|

|2.4+j0.0|

Destructor re = 2.4 im = 0.0

Destructor re = 4.2 im = 5.3

Destructor for d is called before c. since c is constructed first and then d.

Constructor for Automatic Array of Objects

```
class complex
{   private : re_, im_;
    public:
        complex(double re=0.0,
        double im=0.0) : re_(re), im_
        (im)
        {
            cout<<"(constr" <<"re="<<re_
            <<"im="<<im_<<endl;
        }
        ~complex()
        {
            cout<<" Destruct "<<"re="
            <<re_<<" im="<<im_<<
            endl;
        }
        void op(complex(double i)
        {   re_+=i, im_+=i;
        }
        void print()
        {   cout<<"re="<<re_
            <<" im="<<im_<<endl;
        }
};
int main()
{   complex c[3] // constructors called
                 thrice c[0], c[1]
                 c[2]
```

```
for (i=0; i<=2; i++)
{
    c[i].objcomplex(i);
    c[i].print();
}
return 0;
} // scope over so first destructor of
c[2] is called then c[1] and then c[0].
```

If our class has array of objects and then it must support default constructor if not default then it is called require parameter to parameter to be passed. So above 3 calls so 6 parameters to Constructor not possible.

O/p:

```
Constr: re = 0.0, im = 0.0
constr: re = 0.0, im = 0.0
constr: re = 0.0  im = 0.0
re = 0.0, im = 0.0
re = 1.0, im = 1.0
re = 2.0, im = 2.0
Destruct: re = 2.0, im = 2.0   → destructor for c[2]
Destruct: re = 1.0, im = 1.0   → destructor for c[1]
Destru: re = 0.0  im = 0.0
```

↓

destructor for c[0]