(10) Object lifetime : Static (global static object)

```
class complex
{   same as above

};
complex c(4.2, 5.3) // static (global)
```

Object constructed before main starts and destructor after main ends.

```
int main()
{   cout<<" main() starts"<<endl;
    complex d(2.4);
    c.print();
    d.print();
    return 0;
};
```

In C computation has to starts from main But in c++ main still is an entry point in which c is called first by the system. but computation does not necessarily starts with main. computation starts with constructor of static (global) Object which has to be initialized before main starts

* All static Object get constructed before main starts

o/p: { call for static global object
Constructor : re = 4.2, im = 5.3 // for main() starts

Constructor // Constructor : re = 2.4, im = 0.0
call for local object

|4.2 + j 5.3|

|2.4 + j 5.3|

Destructor : re = 2.4, im = 0.0

main ends // Destructor : re = 4.2, im = 5.3

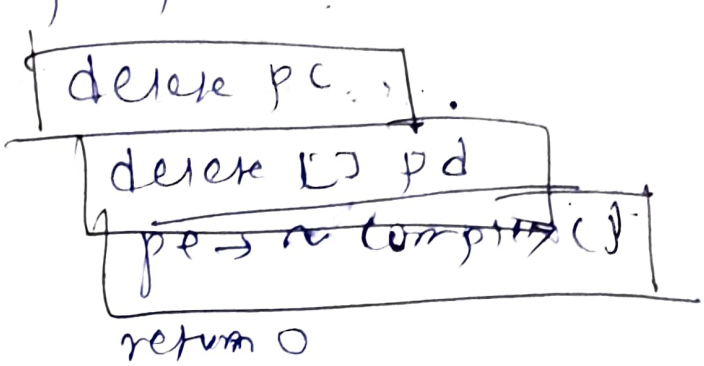
destructor for static (global) object.

Object creation for dynamic object

```
class Complex
{
    some as above;
};
int main()
{
    unsigned char buf[100];
    Complex *pc = new Complex(4.2, 5.3);
    Complex *pd = new Complex[2];
    Complex *pc = new(buf) Complex(2.6, 3.9);
```

Operator new: allocate memory and then @ calls constructor but Constructor is implicit.

pc->print()
pd[0].print();
pd[1].print();



}

* new allocates memory and then calls the corresponding constructor
* array new → new[] allocates memory and then calls default constructor

* Operator placement new : only calls the constructor, there is no allocation of mem(Heap). It uses buf.
* Operator delete: calls the Destructor and then de-allocate the mem.
* Operator delete [] // calls destructor and then release mem.
* For Object created using placement new we have to call the destructor explicitly since there is no mem allocated from Heap.

O/P:

Constructor : re = 4.2, im = 5.3

Constructor : re = 0.0, im = 0.0

Constructor : re = 0.0, im = 0.0

Constructor : re = 2.6, im = 3.9

| 4.2 + j 5.3 |

| 0.0 + j 0.0 |

| 0.0 + j 0.0 |

| 2.6 + j 3.9 |

Destructor : re = 4.2, im = 5.3

pd[1] // Destructor re = 0.0, im = 0.0

pd[0] // Destructor re = 0.0 im = 0.0

Destructor re = 2.6, im = 3.9

NOTE: mem for an object must be available before its construction and can only be released only after its destruction.

order of initialization → order of data members

* order of initialization does not depend on the order in the initialization list. It depends on the order of data members in the definition.

ex:

```
int init_m1(int m)
{ cout<<"Init_m1: "<<m<<endl;
  return m;
}
int init_m2(int m)
{ cout<<"Init_m2: "<<m<<
  endl;
  return m;
}

class X
{ int m1_; // Initialized First
  int m2_; // Initialized 2nd

public:
  X(int m1, int m2): m1_(
  init_m1(m1)), m2_(init_m2(m2))
  { cout<<"Constructor"<<
    endl;
  }
  ~X()
  { cout<<"Destructor"<<endl;
  }
};
```

```
int main()
{
    X a(2,3);
    return 0;
}
```

O/P:

```
init_m1: 2
init_m2: 3
Constructor
Destructor
```

Expn 2

```
class X
{   int m2_;
    int m1_;
public:
    X(int m1, int m2): m1_(init_m1(m1))
      , m2_(init_m2(m2));
    {
        cout<<" Constructor "<<endl;
    }
    ~X()
    {
        cout<<" Destructor "<<endl;
    }
};
int main()
{
    X a(2,3);
}
```

O/P:

init. m2 : 3
, init. m1 : 2

constructor
destructor.

Note: Order of data members can be critical and while writing initialization list we should be aware of how data members are already created.

Exp ex-2,1 from Slides

Q. Lifetime of data members or embedded Object:

ex. class point
{ int x-;
int y-;
public:
point(int x, int y) : x-(x
, y-(y).
cout<<" point
{ point constr" <<endl; }

~point()
{ cout<<" point destr"
<<endl;
}
void print()
{ cout<<"("<<x-<<
", "<<y-<<")";
}
};

```
class Rect
{   Point TL_
    Point BR_
public:
    Rect(int tlx, int tly, int
    brx, int bry): TL_(tlx, tly),
    BR_(brx, bry)
    { cout<<" Rect const"<<endl;
      print()<<endl;
    }
    void print()
    {
      cout<<"[";
      TL_print();
      cout<<" ";
      BR_print();
      cout<<"]";
    }
    ~Rect()
    { cout<<"Rect Destructor";
      print()<<endl; }
};
int main()
{   Rect r(0,2,5,7);
    cout<<endl;
    r.print(); cout<<endl;
    cout<<endl;
}
```

Here destructor will storm

O/P:

Point constr: (0, 2)

Point constr: (5, 7)

Rect constr: [(0, 2) (5, 7)] →

[(0, 2), (5, 7)]

Rect Destructor: [(0, 2) (5, 7)]

Point Destructor: (5, 7)

Point Destr: (0, 2)

destructor for BR_ is called first and then TL_ destructor coz it is constructed first and then BR_

Note: When data members are of user defined type (class type) they are constructed in the order in which they are listed in class and destructed in reverse order of listing.

Next example from slides

Copy constructor

(1) Complex C1 = {4.2, 5.9}; or

C1 (4.2, 5.9)

↓

invokes

Constructor Complex :: Complex (double, double);

(110) which constructor is invoked for following.

```
Complex c2(c1);
Complex c2=c1
```

OR

// using complex object c1 to initialize another object c2

↓

It is the copy constructor that takes another object of the same class and constructs a copy

```
Complex::Complex(const Complex &);
```

↓

It takes another Object as parameter

ex.

```
class Complex
{   private: double re_, im_;

    public:
        Complex(double re, double im)
        : re_(re), im_(im)
        { cout<<"Normal Constructor";
          print();
        }
        Complex (const &Complex &c);
        : re_(c.re_), im_(c.im_)
        {
          cout<<"Copy Constructor:"<<
              endl;
          print();
        }
```

(iii)
```
~complex()
{ cout<<" Destructor :"
    print();
}
void print()
{ cout<<"re: "<<re_<<" im: "<<
        im_<<endl;
}
};
int main()
{ complex c1(4.2, 5.3);
  c2(c1), c3 = c2;
```
↑ copy constructor called

```
c1.print(); c2.print(); c3.print();
}
```

o/p: Normal constructor: 4.2 + j5.2 c1
copy " : 4.2 + j5.3 — c2
" " : 4.2 + j5.3 — c3

|4.2+j5.3| — c1
|4.2+j5.3| — c2
|4.2+j5.3| — c3

Destructor: 4.2+j5.3 — c3
" : " — c2
" : " " — c1

(112) Why do we need copy constructor

* Function call mechanism in C++

(1) Call by reference:

↓

Set the reference of actual parameter as formal parameter. Both the formal parameter and actual parameter store the same location (Object)

(II) Return by reference: set a reference to the computed value as a return value. Both the computed value and the return value store the same location (Object)

(III) Call by value: make a copy (clone) of actual parameter as a formal parameter. This needs a copy constructor.

(IV) Return by value: make a copy of (clone) of computed value as a return value this needs copy constructor

* Copy constructor is needed for initializing the para member of a UDT from a existing value.

* BitCopy for predefined type.

Note: If a user defined type does not have or is not provided with copy constructor then the consequence is Object of that class cannot be passed as call by value mechanism to any fn and same for return by value.

(11) If we want to copy a value of user defined type as a data member then we will face the same situation as call by value situation.

Note: Copy constructor is necessary for

(I) call by value mechanism
(II) return by value "
(III) To initialize an object with other object

ex.1

```
class complex
{
    double re_, im_;
public:
    complex(double re, double im):
    re_(re), im_(im)
    { cout<<" Normal constructor"<<
                endl;
      print();
    }
    complex(const &complex c):
    re_(c.re_), im_(c.im_)
    {
      cout<<"copy constructor :"<<
        print();
    }
    ~complex()
    { cout<<" Destructor :";
      print();
    }
    void print()
    { cout<<"re :"<<re<<"im="<<
                im<<endl;
    }
};
```

```
void Display (complex c_param)
{
    cout<<"In Display:"<<endl;
    c_param.print();
}
int main()
{
    complex c(4.2,5.3);
    Display(c);
}
```

Before the fn Display is called the actual parameter c has to copy to formal parameter c_param and this process is the copy construction process. So at this point copy constructor will be invoked.

→ copy constructor are called to copy c to c_param.

O/p: Normal constructor: 4.2 + j5.3

" copy " : 4.2 + j5.3

Display : 4.2 + j5.3.

Destructor: 4.2 + j5.3 // As we are exiting from Display fn so destructor c_param is called

Destructor: 4.2 + j5.3 // exit from main

If we don't provide copy constructor then this message will not come as compiler provide free copy constructor.

(11.5) Signatures of a copy constructor

myclass (const myclass & Other)

myclass (myclass &other)

myclass (volatile const myclass & Other)

myclass (volatile myclass &other)

↳ used in embedded system

None of the following are copy constructor though they can copy?

myclass(myclass * Other)

myclass (const myclass * Other)

↳ they are normal constructor. they can not be called at the time of call by value

why parameter of copy constructor must be passed as call by reference?

call val

myclass(myclass Other)

↳ The above is an infinite loop as the call to copy constructor itself needs to make copy for the call by value mechanism.

Free copy constructor:

* If no copy constructor is provided by the user the compiler supplies a free copy constructor
* compiler provided copy constructor can not initialize the Object to proper values. It has no code in its body. It performs a bit copy. - i, e it takes a complete bit pattern of Object and makes another copy.

* Bit copy is not same as Object copy..

Userdefined copy constructor(): String.

```
class string:
{   public: char *str_, size_t len_;
    String(char *s) : str_(Strdup(s),
                len_(strlen(str_)) {}

// String ( const String &s): str_(
    strdup(s.str_), strlen(s.
    len_(s.len_)) {}

~String { free(str_); }

void print()
{ cout<<"("<<str_<<":"<<len_
    <<")"<<endl;
}
};
```

(17) void strToUpper (String a)

{ for(int i=0; i<a.len-; i++)

{ a.str_[i] = toupper(a.str_[i]);

~~COUT<<~~

a.print2();

}

}

int main()

{ String s = "Partha"; →

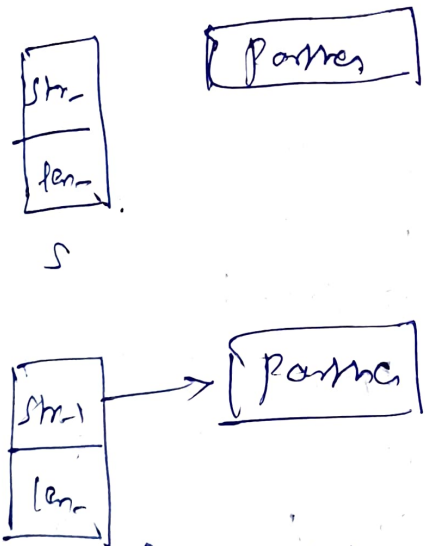
s.print1();

strToUpper(s); o/p: (Partha: 6)

s.print1(); → (PARTHA: 6)

return 0; (Partha: 6)

}



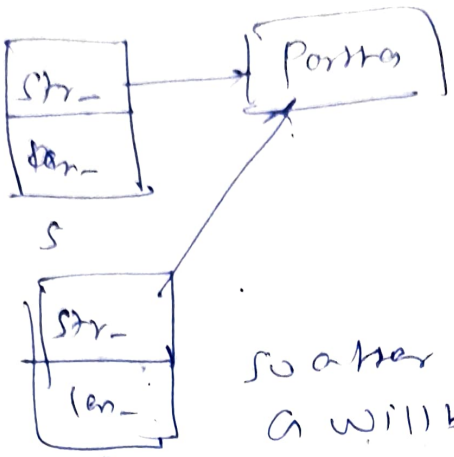
for a object str_ Partha will be ~~co~~ created at new location

Suppose if we remove ~~Cop~~ user defined copy constructor from above code and run then cop: compiler will do bit copy & i.e address will be copy with a's and

both stack pointers to same 'Partha'

like

without user defined copy constructor



so after top upper curly Object a will be destroyed and "Partha" will no more valid. and pgm will crash

O/P: (partha:6)

(PARTHA:6)

(?????):6

some garbage value

Note: when an object has pointer i.e which is referring to another object that are dynamically created then while copying it if we just copy the pointer but do not copy the pointed object then we say we are doing shallow copy. But while copying the object if we do not copy the pointer but we copy the pointed object then we are doing deep copy

copy assignment

copy loosely means clone but in C++ copy doesn't mean clone it means whatever we copy, we can copy one data member or more.

```
class complex
{   double re_, im_;
public:
    complex(double re, double im):
    re_(re), im_(im)
    { cout<<"Normal constructor"<<endl;
      print();
    }
    complex(const complex &c):
    re_(c.re_), im_(c.im_)
    { cout<<"copy constructor:";
      print()
    }
    ~complex()
    { cout<<"Destructor"<<endl;
      print();
    }
    complex & operator=(const complex &c)
    { re_ = c.re_, im_ = c.im_;
      cout<<" copy assignment:";
      print(); return *this;
    }
    void print()
    { cout<<"|"<<re_<<"+i"<<im_<<endl;
    }
};
```

That reference may we require copy constructor

copy assignment

return type is & to make chain assignment possible.

```
int main()
{
    complex c1(4.2, 5.3), c2(7.9, 8.5);
    complex c3(c2);   (copy constructor)
    c1.print(); c2.print(); c3.print();
    c2 = c1;  or  c2.operator=(c1)
    c1 = c2 = c3;
    c1.print();
    c2.print();
    c3.print();
}
```

c1 = (c2 = c3)

Object was not earlier created. Here, we are creating the object c3 and assigning with copy with c2

Objects already created. Here we are assigning value to already created objects.

OP:

Normal constructor : |4.2 + j5.3|

Mor " : |7.9 + j8.5|

copy " : |7.9 + j8.5|

|4.2 + j5.3|
|7.9 + j8.5|
|7.9 + j8.5|

Copy assignment: |4.2 + j5.3|

copy " |7.9 + j8.5| c2 ← c3

copy " |7.9 + j8.5| c1 ← c2

(121)

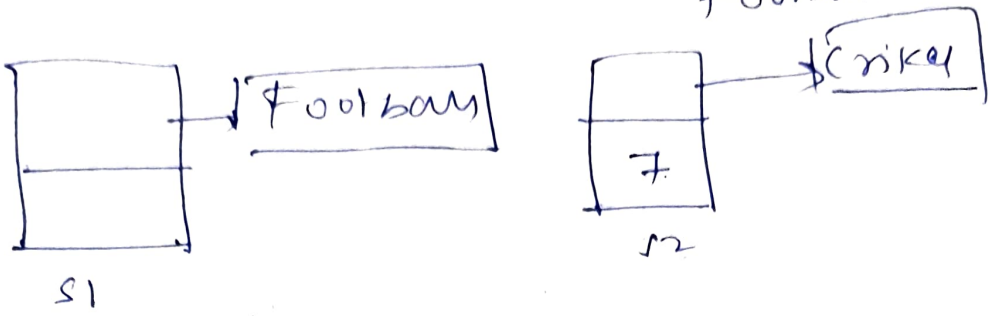
|7.9+j8.5|
|7.9+j8.5|
|7.9+j8.5|

example. copy Assignment

```
class String
{  public: char *str_; size_t len_;
   String(char *s): str_(strdup(s)),
      len_(strlen(str_)) { }
   String & (const String &s):
      str_ (strdup(s.str_), len_(s.len_)
   { }
   ~String()
   { free(str_); }
   String & operator=(const String &s)
   {  free(str_);
      str_ = strdup(s.str_) // perform deepcopy
      len_ = s.len_
      return *this // for chain assignment
   }
   void print()
   {  cout << "(" << str_ << ":" <<
      len_ << ")" << endl;
   }
};
int main()
{  String s1 = "Football", s2 =
   "cricket";
```

```
    s1.print(); s2.print();
    s2 = s1; s2.print();
    return 0;
}
```

Output:
```
Football
Cricket
Football
```



we should do

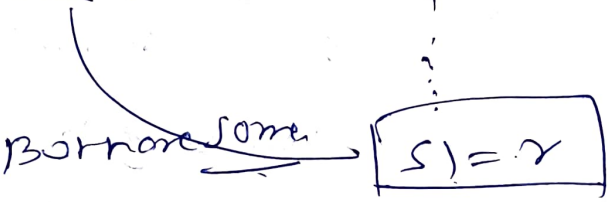
str_ = s.str_ bcoz

(i) Resource held by str_ will leak.

(ii) shallow copy will result with ib related issue

when it we do selfcopy

s1 = s1 or String &r = s1



we have in copy copy assignment.

```
String &( const String &s)
{
    free(str_);
    str_ = strdup(s.str_);
    len_ = s.len_
    return *this
}
```

(123) O/P:

(Football:8)

(Cricket:7)

(????:8) since memory is release in free for (str_) it will be garbage value. so self copy must be protected

Now modified above code

```
String &.(const String &s)
{
    if(this!=&s)
    {
        free(str_);
        str_ = strdup(s.str_);
        len_ = s.len_;
    }
    return *this;
}
```

Note: if the user do not provide a copy assignment operator but uses it in the pgm. then compiler will provide a free copy assignment operator which again just do a bitwise copy without considering specific req.

signature of copy assignment

① `myClass & operator=(const myClass &s)`

```
{
    if(this!=&s)
    // Release resource held by *this
    // copy members of s to members of *this
```

return *this;

}

Signature of copy assignment operator

can be one of

myclass & operator = (const myclass &)

myclass & operator = (myclass &)

following signature also occasionally used

(i) myclass & operator = (myclass rhs)

(ii) const myclass & operator = (const myclass & rhs)

(iii) const myclass & operator = (myclass rhs)

(iv) const myclass & operator = (Myclass rhs)

(v) myclass operator = (const myclass & rhs)

(vi) myclass operator = (myclass & rhs)

(vii) myclass operator = (myclass rhs)

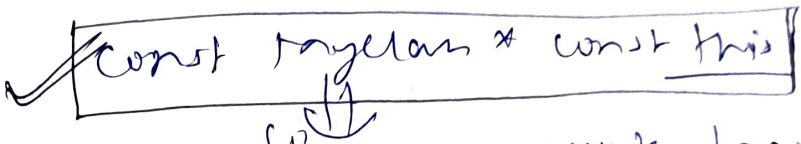
(25) Constant Object

- Like Objects of built in type, Objects of user-defined types can also be made constant

- If an object is constant, none of its data members ~~it can be changed~~

- If an object is constant then its this pointer will become constant type

- The type of this pointer for a constant object of class myclass



constant pointer to constant Object

Instead of

~~Otherwise~~

~~const~~

myclass * const this as for a non constant Object of the same class

- A constant object cannot invoke normal methods (non constant) methods of the class ~~as~~ as this method change the Object.