

Object lifetime: Static (global static)

object

class complex

↳ same as above

7:

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;

8:

~~In C computation starts from main
But in C++ main still is an entry point
in which cor 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 objects get constructed
before main starts

OOP: (called for static global object (102))
Constructor: $r = 4.2$, $m = 5.3$ // For
main() starts

Constructor // Constructor: $r = 2.4$, $m = 0.0$
main() for
d (local
object) ~~constructor~~
 $\frac{d}{|}$
 $| 4.2 + j 5.3 |$
 $| 2.4 + j 5.3 |$

main ends // Destructor: $r = 2.4$, $m = 0.0$
Destructor: $r = 4.2$, $m = 5.3$

Destructor for static (global) object.

Object lifetime for dynamic object

class Complex

↳ some as above:

↳ operator new allocates memory
and then @ calls constructor
But constructor is implicit.

↳ unassigned later but [loop]:

Complex *pc = new Complex(4.2, 5.3)

Complex *pd = new Complex[2]

Complex *pc = new (buf) Complex
(2.6, 3.9);

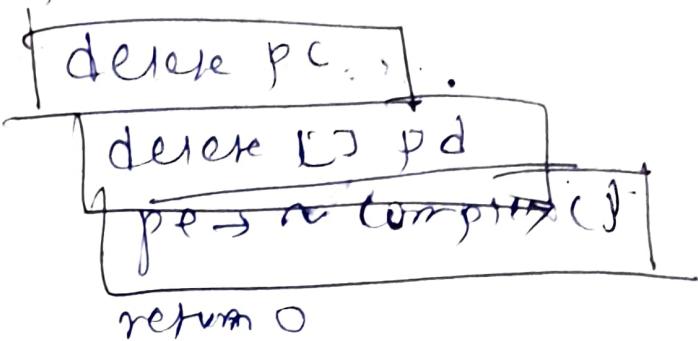
↳ $p \rightarrow \text{ptr}(1)$

$pd[0] \rightarrow \text{ptr}(1)$

$pd[1] \rightarrow \text{ptr}(2)$

(103)

pi → pmt()



Y

* no new allocates memory and
for calls the corresponding constructor

(for

* array new } → all calls mem
[] new and for 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 deallocate the mem.

* operator delete [] // calls destructor
and then release mem

* For objects created using placement new we have to call the destructor
explicitly since there is no mem
allocated from Heap.

Q19:

(104)

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.

(105)

Order of initialization \rightarrow order of

public members

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

Ex: int init_m(int m)

{ cout << "init_m: " << 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 second

public:

X(int m1, int m2): m1_(
init_m1(m1)), m2_(init_m2(m2))

{ cout << "constructor" <<
endl;

}

$\sim X()$

{ cout << "Destructor" << endl;

}

int main()

(106)

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

O/P: init_m1: 2

init_m2: 3

constructor

Destructor

Expt 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

(107)

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 actually treated.

Let's see from slides

Ex-2, 3

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~~ contr<< endl; }

~point()

{ cout << "Point Destor"
<< endl;

void print

{ cout << " " << x <<
" , " << y << endl; }

};

}

4. Point TL-

~~Point BR-~~

public:

Rect (int f1x, bry int f1y, int
bx, bry int bx): TL-(f1x, f1y),
BR-(bx, by)

{ cout << "Rect constructor"; }

private (cond):

}

void print()

{ cout << "L";

TL.print();

cout << " ";

BR.print();

cout << "R" ;

}; (nRect)
(~Rect Destru^o)
int main()

4. Rect r(0,2,5,7);

Here destr^o
will stem

cout << endl;

r.print(); cout << endl;

cout << endl;

};

(109)

Q1:

point constr: $(0, 2)$

point constr: $(5, 7)$

Rect constr: $[(0, 2) (5, 7)] \rightarrow$

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

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

Point Destructor: $(5, 7)$

Point Destr: $(0, 2)$

Destructor for BR_{-} is called first
and then TL_{-} destructor gets called
 BR_{-} 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 destroyed in the reverse order

of listing:

Next example from slides

Copy constructor

(10) complex c1 = {4.2, 5.9}; or

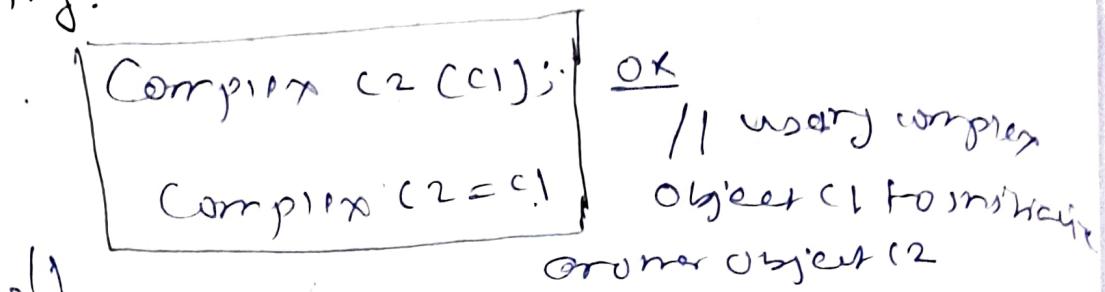
c1(4.2, 5.9)

↓

invokes

Constructor complex:: complex(
double, double);

(110) which constructor is invoked for following.



Complex::Complex(const Complex &);
↓
It forces another object as
parameter

Ex. class Complex

h. private: double re_, im_;

public:

Complex(double re, double im)
: re_(re), im_(im)

h. cout<<"Normal constructor"
ptr);
}

Complex(const Complex &c):
re_(c.re_), im_(c.im_)
~~re_=c.re, im_=c.im~~

h. cout<<"Copy constructor:"<<
endl;
ptr);
}

```

③ ncomplex()
{
    cout << "Destructor : " cout
    ~print();
}

void print()
{
    cout << re << " + " << im << endl;
}
int main()
{
    complex c1(4.2, 5.3);
    c2 = c1 // copy constructor
    c1.print(); c2.print(); c3.print();
}

```

Q1:

Normal constructor: $4.2 + j5.3 \rightarrow c1$

copy		$4.2 + j5.3 \rightarrow c2$
		$4.2 + j5.3 \rightarrow c3$

$(4.2 + j5.3) \rightarrow c1$

$(4.2 + j5.3) \rightarrow c2$

$4.2 + j5.3 \rightarrow c3$

Destructor: $4.2 + j5.3 \rightarrow c3$

	:		$\rightarrow c2$
	:		$\rightarrow c1$

(ii)

why do you need copy constructor

* function call mechanism in C++

(i) call by reference:

↓
set the reference of actual parameters as formal parameter. Both the formal parameter and actual parameters share 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 share the same location (Object)

(iii) call by value: make a copy (clone) of actual parameters as a formal parameter. This needs a copy constructor ↗

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

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

* Bit copy for predefined type.

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

If we want to copy a value of user defined type as a data member then we will face some situation as copy by value situation.

Note: Copy constructor is necessary

for

call by value mechanism

return by value "

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;
};

Complex(const & complex c) :

{ re_=c.re_, im_=c.im_ }

{

cout << "Copy. constructor : " ;
endl;

};

cout << "Destructor : " ;

endl;

void print()

{ cout << "re : " << re_ << "im : " <<

im_ ; endl;

```
void Display (complex c-param) (114)
```

```
{ cout << "Display:" <<
```

c-param

```
c-param.print();
```

}

```
int main()
```

```
{ complex C(4.2,5.3);
```

Display C ->

Before the fn Display
returns, is called the current

parameter C has to copy to formal
parameter c-param, and this process is the
copy construction process. So overrider's
copy constructor will be invoked.

Copy constructor are called to copy C to
c-param.

Q19: Normal constructor: $4.2 + j5.3$

// copy : $4.2 + j5.3$

If we don't provide copy constructor:

provide copy destructor: $4.2 + j5.3$ // As we are
exiting from Display fn so

destructor of c-param is called
will not come

as final destructor: $4.2 + j5.3$ if exit from
provide free copy constructor.

(115) 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 calling value.

why parameters of copy constructor must be passed as copy by reference?

myclass (myclass & Other)

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

Free copy constructor

(116)

- * If no copy constructor is provided by the user the compiler supplies a free copy constructor.
- * Compiler provided copy constructor cannot initialize the object to proper values. It has no code in its body. It performs a bit copy - i.e. it makes a complete bit pattern of object and makes another copy.
- * Bit copy is not same as object copy.

User defined 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_ ), the string( s.  
    len_( s.len_ ) ) {}
```

```
~string { free(str_); }
```

void print()

```
{ cout << " " << str_ << " " << len_<< endl;
```

?;

12) void strToUppper (String a)

L for (int i=0; i < a.length(); i++)

L. a.str[i] = toupper(a.str[i]);

~~cout <<~~

a.print();

}
y
y

int main()

A String s = "Portia"; ↗

s.print();

strToUppper(s); ↗ op: (Portia: 6)

s.print(); ↗ (PARKIA: 6)

return 0; ↗

(Portia: 6)

}



[Portia]

After toupper a
will be destroyed



A. for a object str - Portia will be created at new location

Suppose if we remove copy user defined

copy constructor from above code and run

then copy compiler will do bit copy

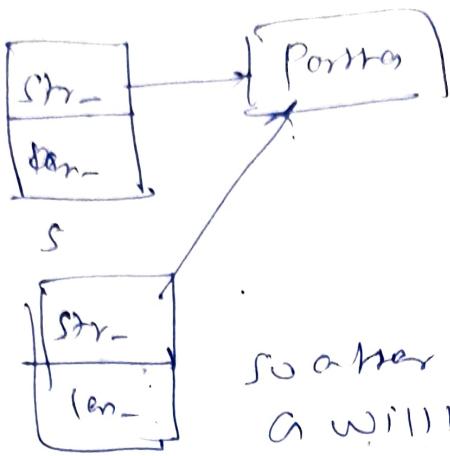
* i.e address will be copy with original

both starts pointing to same shallow 'Partha'

else

without user defined copy constructor

(ii)



so after toupper call Obj
a will be destroyed and

'Partha' will no more valid.

and program 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 if we just copy the pointer but don't 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

(119)

~~copy~~ loosely means clone. But in C++
~~copy~~ does not mean clone. It means whatever
~~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"
            << endl;
        print();
    }
}
```

Not reference ~complex()

May we require ~complex()
copy constructor

copy assignment if

Complex & operator=

Complex const (Complex &c) &

return type is &
to make chain
different possible.

re_ = c.re_, im_ = c.im_;

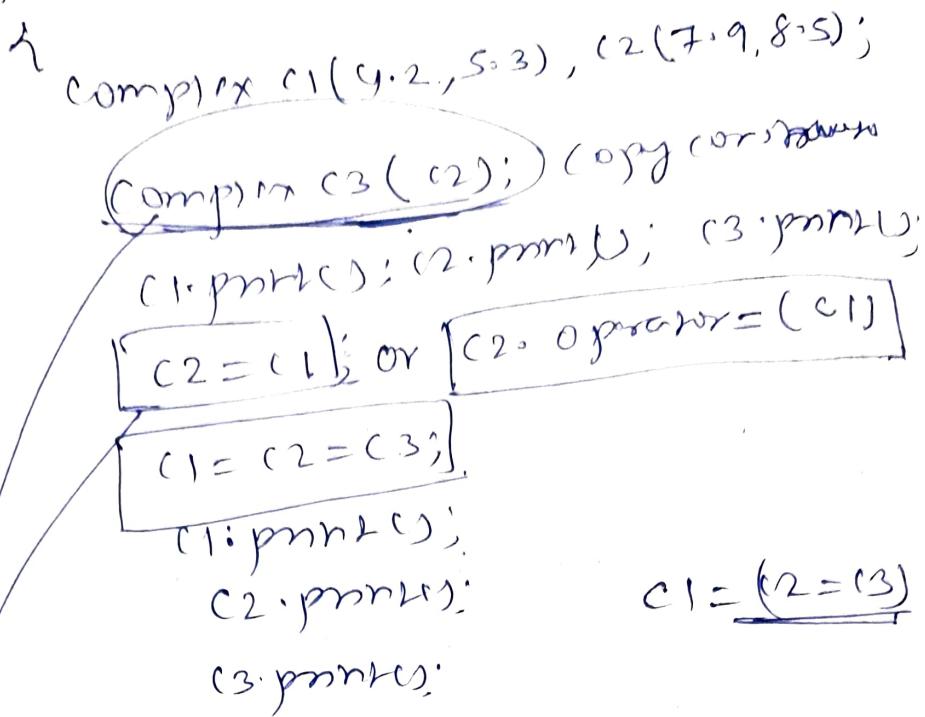
cout << "copy assignment"
 << endl;
 print();
 return *this;

void print()

cout << "re_ = " << re_
 << endl;
 cout << "im_ = " << im_
 << endl;

int main()

(120)



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

→ Object is already created. Now we are assigning value to already created object.

OP: Normal constructor: | 4.2 + j 5.3 |

More " | 7.9 + j 8.5 |

Copy " : | 7.9 + j 8.5 |

| 4.2 + j 5.3 |

| 7.9 + j 8.5 |

| 7.9 + j 8.5 |

Copy assignment: | @ 4.2 + j 5.3 |

Copy " | 7.9 + j 8.5 | ← c2 ← c3

copy " | 7.9 + j 8.5 | ← c1 ← c2

(12)

$$\begin{array}{|c|} \hline 7.9 + j8.5 \\ \hline 7.9 + j8.5 \\ \hline 7.9 + j8.5 \\ \hline \end{array}$$

example: copy Assignment

class String

public: char *str_; size_t len_;

String(char *s): str_(std::string(s)),

len_(strlen(str_)) {}

String & (const String &):

str_ = std::string(s.str_), len_ = s.len_;

{} }

~String()

{ free(str_); }

String & operator=(const String &)

{ free(str_);

str_ = std::string(s.str_); // perform

len_ = s.len_;

deep copy.

return *this; // for chain

assignment

void print()

{ cout << "(" << str_ << ":" <<

len_ << ")" << endl;

}

};

int main()

{ String s1 = "Football", s2 =

"Cricket";

s1.print(); s2.print();

(122)

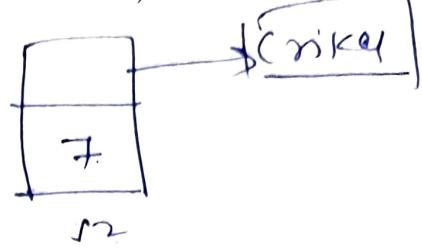
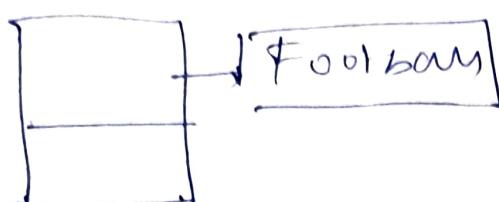
s2 = s1; s2.print();

return 0; O/P:

(Football:S)

(Cricket:7)

Football:8



we should not do

str = ~~s~~ str - bcz

① Resource held by str will leak.

~~it~~ shallow copy will result with ib related issue.

when if we do selfcopy

| s1 = s1 or str = &r = s1

Borrow some | s1 = r

we have in copy copy assignment.

String & (const ~~String &~~)

L free(str);

str = strdup(s.str);

len = s.len

return *this

(Football:8)

(Cricket:7)

(PPP:8) since memory is released in free for str- it will be garbage value. so self copy must be protected

Now modified above code

String 2. (const String 2)

{ ~~char* str=~~

 if (*this != &s)

 { free(str-);

 str = strdup(s->str-);

 len- = s->len-;

 } ~~return~~

 }

~~Note if the user does not provide a copy assignment operator but uses it in the program 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 2. Operator (const myClass &s)

{ if (*this) = &s)

 // Release resource held by ~~*this~~

 // Copy members of s to members of *this

return & hrs

(124)

Signature of copy assignment operator

can be one of

~~myclass & operator = (const myclass
&)~~

myclass & operator = (myclass &)

following signature ~~occurs~~ occasionally
used

- ① myclass & operator= myclass &rhs)
- ② const myclass & operator = (const
myclass &rhs)
- ③ const myclass & operator = (my
class &rhs)
- ④ const myclass & operator = (Myclass
&rhs)
- ⑤ myclass operator = (const myclass
&rhs)
- ⑥ myclass operator = (myclass &rhs)
- ⑦ myclass operator = (myclass &rhs)

const Object

Like objects of built-in types, objects of user-defined types can also be made constant.

If an object is constant, none of its data members can be changed.

If an object is constant having this pointer will become constant type.

The type of this pointer for a constant object of class Myclass

const Myclass * const this

With

constant pointer to const

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. Test this
method change the object.