

Contents

1	lecture 02 06/04/19	2
1.1	inline function	2
1.2	static members	2
1.3	scope resolution operator	2
2	lecture 03 06/06/19	3
2.1	member initialization list	3
2.2	Redifining	3
2.3	constructors	3
2.4	OOD (object oriented design) fundamentals	3
2.5	Access levels	3
3	lecture 04 06/10/19	4
3.1	Operator Overloading	4
3.1.1	overloading example	4
4	lecture 05 06/11/19	5
4.1	Operator overloading contd.	5
5	lecture 06 06/12/19	6
5.1	Pointer and Reference review	6
6	lecture 07 06/13/19	7
6.1	Pointers and Dynamic variables	7
6.2	copy constructor	7
7	lecture 08 06/17/19	8
7.1	Copy Constructor for derived class Example	8
8	lecture 09 06/18/19	9
8.1	vector copy constructor example	9
8.2	copy assignment	9
8.3	recursion	9

1 lecture 02 06/04/19

OOP-review:

1.1 inline function

member function definition given completely in the definition of the class saves overhead of a function invocation very short definitions

1.2 static members

keyword static is used, global variable or member static member functions can be accessed without an object ever being created `class::memberFunction()`

private: static int y; //will be shared by all object instances

1.3 scope resolution operator

::

2 lecture 03 06/06/19

OOP-review cont:

2.1 member initialization list

```
member initialization list for base class  
using base class constructor
```

- Cat(int a, string b, bool c): Animal(d, e, f)

2.2 Redifining

overloading - same name but different parameters, usually occurs in same class, fn, etc. overriding - same function signature/prototype, inheritance is usually involved

2.3 constructors

derived class constructor can't access private base class data, must call base class constructor in deriv.

2.4 OOD (object oriented design) fundamentals

- encapsulation
- inheritance
- polymorphism

ex) pShape->draw();

Shape is a pointer of base class and can point to Circle obj or Square or etc.. each have different virtual draw

2.5 Access levels

- public
- protected
- private

3 lecture 04 06/10/19

3.1 Operator Overloading

- most existing **not scope resolution or member access** C++ operators can be overloaded
- New operators cannot be created
- an operator function is a function that overloads an operator

binary operator with two operands

```
Deck a,b;
```

```
bool isEqual a == b
```

a.operator==(b) same as a == b

3.1.1 overloading example

```
bool operator<=(const clockType& otherClock const);
```

```
^ otherClock is being passed in  
as if (clock <= otherClock) rhs  
operator always passed in  
with lhs considered as invoking  
object
```

4 lecture 05 06/11/19

4.1 Operator overloading contd.

Pre and post inc

`++c` **vs** `c++`

- Pre has slightly less overhead and `++` happens before assignment
- `++` is a unary operation ***one*** operand

IC exersize

```
clockType clockType::operator(int x)
{
    clockType temp = *this; // this is a copy operation using copy constructor
    /* increment code */
    return *temp; // will return original clock value but still increment
                  // the operand
}
```

5 lecture 06 06/12/19

5.1 Pointer and Reference review

```
int count = 100;           // initialized on the stack
int* pCount = nullptr      // same as NULL;

pCount = &count;           // pointer set to the address of count
Clock* pClock = new Clock(); // allocates on the heap and returns a
                             // pointer which is assigned to pClock

void* voidPtr; //can be used to point to any type

std::cout << pCount;      // returns address pointer is pointing to
std::cout << *pCount;     // returns data pointer is pointing to
```

in reality a reference is a **specialized const pointer**

```
int& rCount = count; // If declaring a reference ,
                    // must say what it refers to
```

a reference can be used **interchangably** with the object its self

```
std::cout << &rCount; // will output address of object rCount
                      // refers to, in this case the address
                      // of count
```

6 lecture 07 06/13/19

6.1 Pointers and Dynamic variables

```
int *p;
p = new int [10]

*p = 25; // stores 25 in first mem location
p++;    // moves pointer to next array component
*p = 35; // sets next array component to 35
```

6.2 copy constructor

```
/* both call copy constructor */
ptrM objB = objA;
ptrM objB(objA);
```

shallow copy (*default copy constructor*) **will not work** if object contains pointers that point to data such as the array on heap above.

deep copy constructor *makes complete copy of object*, can allocate new array on heap

```
/* deep copy constructor */
ptrMemVarType::ptrMemVarType(const ptrMemVarType &otherObj)
{
    maxSize = otherObj.size;
    length = otherObj.length;

    p = new int [maxSize];

    for(int i = 0; i < length; i++)
        p[i] = otherObj.p[i];
}
```

7 lecture 08 06/17/19

7.1 Copy Constructor for derived class Example

```
//calling the base class copy constructor in the member init list
CityTempLatitudeLongitude(const CityTempLatitudeLongitude &otherObj) : CityTemp(otherObj)
// shallow copy will work for B-class data (no )
{
    latitude = new float [NUM_ROW]
    longitude = new float [NUM_ROW]
    for (int i = 0; i < NUM_ROW; i++)
    {
        latitude = otherObj.latitude[i]
        longitude = otherObj.longitude[i]
    }
}
```


8 lecture 09 06/18/19

8.1 vector copy constructor example

```
/* pt 1: copy automatic data (not pointed to) first */
vector(const vector &otherObj) : size_v{otherObj.size_v}, elem{new double[otherObj.size]},
    space{otherObj.space}
{
    /* pt 2: dynamically allocate pointed to data (array of doubles) */
    std::copy(otherObj.elem, otherObj.elem + size_v, elem)
}
```

8.2 copy assignment

similar to the copy constructor however information needs to be copied into an existing object

```
vector &operator=(const vector &otherObj)
{
    /* pt 1: release pointed to data which obj has ownership of */

    /* code... */

    /* pt 2: pt1 & 2 from copy constructor */
}
```

8.3 recursion

factorial example:

```
float fact(int n)
{
    //factorial of n = n * (n-1) * (n-2) ... * 1
    return n > 1? n* fact(n-1) : 1;
}
```

Albert:

The Cookie