Contents

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 initailization 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 fuction 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) fundementals

- 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 opperation ***one*** operand

IC exersize

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