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
```

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 alocate 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;
}
```

9 lecture 10 06/19/19

9.1 recursion cotd...

solving a problem by reducing it to a smaller version of its self

constexpr declares a an expresion as const

9.2 polymorphism

pure virtual function used in interface inheritance

9.3 Exam 1

- use **friend** function with mixed types and ex) << and >>
- know order of constructors and destructors called in derived classes
-

10 lecture 11 06/24/19

10.1 Virtual Functions, ABCs and Namespaces

Virtual function - **dynamic** binding occurs at run time, not compile time.

Non virtual functions are bound **statially** at compile time.

- virtual only needs to be declared in base class, automatically virtual in derived.
- Object slicing can occur if passing base class object by value, results in extra derived class data being sliced off (base class copy constructor is called on the derived class object)
- can be avoided be by using references or pointers
- c++11 keyword override can be used to indicate if overiding virtual functions from the base class

  pure virtual and ABC

```
virtual pureVirtual() = 0;
```

- expression is any logical expression
- if true next expression evaluates prgm continues
- if false prgm terminates and indicates where error occured

```
#include <cassert>
assert(expression)
```

namespaces syntax members are variable declarations etc.

```
namespace nsp_name
{
  members
}

//using namespaces

using namespace name_space
/* or */
using name_space::member
```

```
//ec practice inclass

void Rectangle::Print()
{
  cout << l << endl;
  cout << w << endl;
  cout << x << endl;
  cout << y << endl;
}
```

```
void Circle::Print()
{
  cout << l << endl;
  cout << w << endl;
  cout << r << endl;
}

void Rectangle::UpdateDimentions(int l, int w, int r) : l{l}, w{w}
{}

void Circle::UpdateDimentions(int l, int w, int r) : r{r}
{}
```

11 lecture 12 06/25/19

11.1 In class initalizer

```
int r {100} // in-class declaration and initialization
```

11.2 Exception Handling

basics

- try/catch block
- errors are handled in the catch block
- assert keyword (older C style way, not needed)

```
try
{
  //statements
  throw somethingToThrow
}
catch (dataType1 identifier)
{
  //err handling code
}
catch (dataTypeN identifier)
{
  //err handling code
}
catch (...) // catch all, catches any error
{
  //err handling code
}
```

12 lecture 13 06/26/19

Exception Handling contd..

12.1 noexcept specifier

noexcept will guarentee that no exeption can be thrown in a function

```
noexcept(expression); //if evals to true the function is declared not to throw any exceptions
```

12.2 stack unwinding

when an exception is thrown and execution jumps to catch block, automatic variables from try block must be deleted to avoid mem leaks.

12.3 static casting

C-like casting

and

down casting is allowed in static_cast

```
Child *p = static_cast<Child *>(&Parent);
```

12.4 up and down casting

- up cast - moving up in heirarchy (always allowed/safe)
- down cast moving down in heirarchy (not always possible/safe)

12.5 dynamic casting

can occur at runtime must be an up-cast `Child *p = dynamic_cast<Child *>(pParent); //returns pointer if able to perform cast //, if unable returns null ptr (this occurs when trying a down-cast)`

13 lecture 14 06/27/19

13.1 casting contd. . .

casting, both dynamic and static must occur between objects in an inheritance hierarchy

14 lecture 15 07/01/19

- **user-stories due on Friday**

14.1 templates

- enables you to write generic code
- simplify function overloading

synatax

```
template <typename Type>
Type larger(Type a, Type b)
{return (a > b)? a: b;}
```

as a normal function

```
int larger(int a, int b)
{return (a > b)? a: b;}
```

15 lecture 16 08/02/19

Templates are "blueprints" for making function definitions.

template is instantiated when called

```
//calling template fn

int main()
{
  larger<int>(10,5); //template instantiation
}
```

```
//multiple types
template <typename T1, typename T2>
T larger(T1 a, T1 b)
{
  return (a > b)? a: b;
}

larger<int, Clock>(a, b)  //multiple types
```

16 lecture 17 08/03/19

vector template example

```
template <typename T1>
class vector
{
    /*
    vector of doubles much like stl vector container

    NOTE: elem[n] is vector component n for all n >= 0 AND n < size_v
          size_v = the number of items stored in the vector
          space = the available storage capacity of the vector where size_v <= space
          if size_v < space there is space for (space - size_v) doubles after elem[size_v-1]
    */

    int size_v;    // the size
    T1 *elem; // pointer to the elements (or 0)
    int space;     // number of elements plus number of free slots
public:
    vector() : size_v{0}, elem{nullptr}, space{0} {} // default constructor

    explicit vector(int s) : size_v{s}, elem{new T1[s]}, space{s} // alternate constructor
    {
        for (int i = 0; i < size_v; ++i)
            elem[i] = 0; // elements are initialized
    }

    vector(const vector &src) : size_v{src.size_v}, elem{new T1[src.size_v]}, space{src.
    space} // copy constructor
    {
        copy(src.elem, src.elem + size_v, elem); // copy elements - std::copy() algorithm
    }

    vector &operator=(const vector &src) // copy assignment
    {
        T1 *p = new T1[src.size_v];         // allocate new space
        copy(src.elem, src.elem + src.size_v, p); // copy elements - std::copy() algorithm
        delete[] elem;                            // deallocate old space
        elem = p;                                 // now we can reset elem
        size_v = src.size_v;
        return *this;  // return a self-reference
    }

    ~vector() {
        delete[] elem; // destructor
    }

    T1 &operator[](int n) {
        return elem[n]; // access: return reference
    }

    const T1 &operator[](int n) const {
        return elem[n];
    }

    int size() const {
        return size_v;
    }

    int capacity() const {
        return space;
    }
```

```
void resize(int newsize) // growth
// make the vector have newsize elements
// initialize each new element with the default value 0.0
{
    reserve(newsize);
    for (int i = size_v; i < newsize; ++i)
        elem[i] = 0; // initialize new elements
    size_v = newsize;
}

void push_back(T1 d)
// increase vector size by one; initialize the new element with d
{
    if (space == 0)
        reserve(8);          // start with space for 8 elements
    else if (size_v == space)
        reserve(2 * space); // get more space
    elem[size_v] = d;        // add d at end
    ++size_v;                // increase the size (size_v is the number of elements)
}

void reserve(int newalloc)
{
    // never decrease allocation
    // allocate new space

    // copy old elements
    // deallocate old space
}

using iterator = T1 *;
using const_iterator = const T1 *;

iterator begin() // points to first element
{
    if (size_v == 0)
        return nullptr;
    return &elem[0];
}

const_iterator begin() const
{
    if (size_v == 0)
        return nullptr;
    return &elem[0];
}

iterator end() // points to one beyond the last element
{
    if (size_v == 0)
        return nullptr;
    return &elem[size_v];
}

const_iterator end() const
{
    if (size_v == 0)
        return nullptr;
    return &elem[size_v];
}

iterator insert(iterator p, const T1 &val) // insert a new element val before p
{
    // make sure we have space
```

```
        // the place to put value

        // copy element one position to the right
        // insert value

        return nullptr; // temp remove & replace
    }

    iterator erase(iterator p) // remove element pointed to by p
    {
        if (p == end())
            return p;
        for (iterator pos = p + 1; pos != end(); ++pos)
            *(pos - 1) = *pos; // copy element one position to the left
        //delete (end() - 1);
        --size_v;
        return p;
    }
};
```

16.1 Big O

an equation of how the runtime scales with increase in data

- O(1) runtime is non dependent on data size
- O(N) linear, runtime increases linearly with size increase
- O(N^2) squared, runtime increases by n^2 times where n is size
- O(N*A) different data sizes get different variables
- Big O only cares about dominant terms ex) O(X^2 + X) drop X because X^2 is dominant
  - as X gets large the + X becomes insignificant