*lectures 7 (and 8 or 9) (1 of 2 slide decks)*

- *C++ at a Glance - introduction to …*
  - *C++ is an object oriented programming language*
    - *classes, members, methods*
      - *access levels (encapsulation)*
    - *function overloading*
    - *templates*
    - *constructors, destructors*
    - *glimpse of inheritance*
      - *method overriding*

*Kenneth J. Roche*
*University of Washington*
*AMATH 483 / 583 High Performance Scientific Computing*
*Spring Quarter 2025*

# *CLASSES -intro*

```
class rectangle {};
```

Classes

- recipe to create object
  - use keyword ***class***
- fields in the class are variables which describe the state of the object

```
class rectangle
{
    int x, y;
};
```

*Kenneth J. Roche*
*University of Washington*

# CLASSES

*Classes*

- ***methods*** *are functions for the class*

*note the scope resolution: method is prefixed with the class name*

*return type*

```
class rectangle
{
    int x, y;
    int area(); //this is a method declaration
};

int rectangle::area() { return x*y;}
```

*Kenneth J. Roche*
*University of Washington*

## *Classes*

- *access levels*
  - ***public*** *- accessible from anywhere in the program*
  - ***private*** *-accessible only within the class (define public set/access methods)*
  - ***protected*** *-accessible within the class and by derived classes*

*private implicitly selected*

```
class rectangle
{
    int x, y;
};
```

*public explicitly selected*

```
class rectangle
{
public:
    int x, y;
    int area(); // this is a method declaration
};
```

*Kenneth J. Roche*
*University of Washington*

# *CLASSES*

*Classes*

- *declaring and using class object and methods*

```cpp
#include <iostream>

class rectangle
{
public:
    int x, y;
    int area(); // this is a method declaration
};

int rectangle::area() { return x * y; }

int main()
{
    rectangle rect;
    rect.x = 3;
    rect.y = 4;

    std::cout << "area: " << rect.area() << std::endl;
    return 0;
}
```

*Kenneth J. Roche*
*University of Washington*

# *CLASSES*

*Classes*

- *declaring and using class object and methods*

```
int main()
{
    rectangle rect;
    rect.x = 3;
    rect.y = 4;
    std::cout << "area: " << rect.area() << std::endl;

    rectangle r2;
    r2.x = 3;
    r2.y = 5;
    std::cout << "area: " << r2.area() << std::endl;

    return 0;
}
```

*Kenneth J. Roche*
*University of Washington*

```cpp
class AccessDemo
{
public:
    int publicVar;

    void setPrivateVar(int val)
    {
        privateVar = val;
    }

    void setProtectedVar(int val)
    {
        protectedVar = val;
    }

    void display()
    {
        std::cout << "publicVar = " << publicVar << std::endl;
        std::cout << "privateVar = " << privateVar << std::endl;
        std::cout << "protectedVar = " << protectedVar << std::endl;
    }

private:
    int privateVar;

protected:
    int protectedVar;
};
```

```cpp
int main()
{
    AccessDemo obj;

    // Accessing public member directly
    obj.publicVar = 1;
    // the following lines would cause compilation errors due to access restrictions
    // obj.privateVar = 4;    // Error: 'privateVar' is private within this context
    // obj.protectedVar = 5;  // Error: 'protectedVar' is protected within this context

    // Accessing private and protected members via public methods
    obj.setPrivateVar(2);
    obj.setProtectedVar(3);

    obj.display();

    return 0;
}
```

*dot operator to access and set class members, use methods*

```
bash-3.2$ g++ -std=c++17 -c class-access.cpp
bash-3.2$ g++ -o xclass-access class-access.o
bash-3.2$ ./xclass-access
publicVar = 1
privateVar = 2
protectedVar = 3
bash-3.2$
```

*Kenneth J. Roche*
*University of Washington*

# CLASSES

## Classes

- *accessing members with pointers*
- *modifying members with pointers*

```
rectangle r2;
r2.x = 3;
r2.y = 5;
std::cout << "area: " << r2.area() << std::endl;

// use of pointer
rectangle *pr = &r2;
std::cout << "area (pointer use): " << pr->area() << std::endl;

// modify with pointer ...
pr->x = 6;
pr->y = 8;
std::cout << "area (pointer use): " << pr->area() << std::endl;
```

*Kenneth J. Roche*
*University of Washington*

## *Classes*

- *equivalent result, different syntax*
  - *accessing members with pointers*
  - *modifying members with pointers*

```
// modify with pointer ...
pr->x = 6;
pr->y = 8;
std::cout << "area (pointer use): " << pr->area() << std::endl;

// equivalent notation ...
(*pr).x = 5;
(*pr).y = 10;
std::cout << "area (pointer use): " << (*pr).area() << std::endl;
```

```cpp
std::vector<int> vectorSum(
    const std::vector<int> &a,
    const std::vector<int> &b)
{ // simple function for adding integer vector types
    if (a.size() != b.size())
    {
        std::cerr << "Error: vectors have different sizes" << std::endl;
        return std::vector<int>(); // Return an empty vector
    }

    std::vector<int> c(a.size()); // Result vector

    // Perform vector addition
    for (int i = 0; i < a.size(); i++)
    {
        c[i] = a[i] + b[i];
    }

    return c;
}
```

*identical functions but distinct data types*

```cpp
std::vector<double> vectorSum(
    const std::vector<double> &a,
    const std::vector<double> &b)
{
    if (a.size() != b.size())
    {
        std::cerr << "Error: vectors have different sizes" << std::endl;
        return std::vector<double>(); // Return an empty vector
    }

    std::vector<double> c(a.size()); // Result vector

    // Perform vector addition
    for (int i = 0; i < a.size(); i++)
    {
        c[i] = a[i] + b[i];
    }

    return c;
}
```

***overloading*** *functions:*
*both functions are in the same source code file AND have the same name*
*… no problem!*

*Kenneth J. Roche*
*University of Washington*

*template functions - generic programming (simplify)*

- *template keyword*

```
template <typename T>
void func() {
    typename T::value_type val; // 'typename' is necessary here
}
```

- *template type parameter, T*

```
// templates typename
template <typename T>
```

- *typename, class keywords*

```
// templates class
template <class T>
```

*Kenneth J. Roche*
*University of Washington*
*AMATH 483 / 583 High Performance Scientific Computing*
*Spring Quarter 2025*

# *template functions - generic programming (simplify)*

- *typename*
- *class*

```cpp
template <typename T>
std::vector<T> vectorSum(
    const std::vector<T> &a,
    const std::vector<T> &b)
{
    if (a.size() != b.size())
    {
        std::cerr << "Error: vectors have different sizes" << std::endl;
        return std::vector<T>(); // Return an empty vector
    }

    std::vector<T> c(a.size()); // Result vector

    // Perform vector addition
    for (int i = 0; i < a.size(); i++)
    {
        c[i] = a[i] + b[i];
    }

    return c;
}
```

```cpp
template <typename T>
std::vector<T> vectorSum(
    const std::vector<T> &a,
    const std::vector<T> &b)
{
    if (a.size() != b.size())
    {
        std::cerr << "Error: vectors have different sizes" << std::endl;
        return std::vector<T>(); // Return an empty vector
    }

    std::vector<T> c(a.size()); // Result vector

    // Perform vector addition
    for (int i = 0; i < a.size(); i++)
    {
        c[i] = a[i] + b[i];
    }

    return c;
}
```

*template functions - generic programming*

```cpp
// integer
std::vector<int> a = {1, 2, 3};
std::vector<int> d = {4, 5, 6, 7};
std::vector<int> c = vectorSum(a, d); // throw an error
std::vector<int> b = {4, 5, 6};
c = vectorSum(a, b);
```

```cpp
// double
std::vector<double> x = {1.5, 2.0, 3.5};
std::vector<double> y = {4.0, 5.5, 6.0};
std::vector<double> z = vectorSum(x, y);
```

*Kenneth J. Roche*
*University of Washington*

*template functions - generic programming*

- *typename*
- *class*

```
// templates class
void kswp(int &a, int &b)
{
    int tmp = a;
    a = b;
    b = tmp;
}

template <class T>
void kswp(T &a, T &b)
{
    T tmp = a;
    a = b;
    b = tmp;
}
```

*Kenneth J. Roche*
*University of Washington*

# *template functions - generic programming*

```
rectangle r1;
r1.x = 3;
r1.y = 4;
std::cout << "area: " << r1.area() << std::endl;

rectangle r2;
r2.x = 3;
r2.y = 5;
std::cout << "area: " << r2.area() << std::endl;
```

```
class rectangle
{
public:
    int x, y;
    int area(); // method declaration
};
int rectangle::area() { return x * y; }
```

```
// use of template class for swapping class rectangle ...
std::cout << "Before swapping:" << std::endl;
std::cout << "r1.x = " << r1.x << ", r1.y = " << r1.y << std::endl;
std::cout << "r2.x = " << r2.x << ", r2.y = " << r2.y << std::endl;
kswp<rectangle>(r1, r2); // swap r1 and r2
// kswp(r1, r2);
std::cout << "After swapping:" << std::endl;
std::cout << "r1.x = " << r1.x << ", r1.y = " << r1.y << std::endl;
std::cout << "r2.x = " << r2.x << ", r2.y = " << r2.y << std::endl;
```

```
Before swapping:
r1.x = 3, r1.y = 4
r2.x = 5, r2.y = 10
After swapping:
r1.x = 5, r1.y = 10
r2.x = 3, r2.y = 4
```

*Kenneth J. Roche*
*University of Washington*

```cpp
bool S1 = true;
bool S2 = false;
std::cout << "Before swapping boolean:" << std::endl;
std::cout << "S1:" << (S1 == 1 ? "true" : "false") << std::endl;
std::cout << "S2:" << (S2 == 1 ? "true" : "false") << std::endl;

kswp<bool>(S1, S2); // swap S1 and S2

std::cout << "After swapping boolean:" << std::endl;
std::cout << "S1:" << (S1 == 1 ? "true" : "false") << std::endl;
std::cout << "S2:" << (S2 == 1 ? "true" : "false") << std::endl;
```

*template functions - generic programming*

```
Before swapping boolean:
S1:true
S2:false
After swapping boolean:
S1:false
S2:true
```

*template functions - generic programming*

**Implicit Instantiation (Type Deduction):** The compiler deduces the template parameter based on the function arguments.

```
template <class T>
void foo(T value) {
    // ...
}

int main() {
    foo(42); // T is deduced as int
}
```

**Explicit Instantiation:** Programmer specifies the template argument explicitly using angle brackets.

```
template <class T>
void foo(T value) {
    // ...
}

int main() {
    foo<int>(42); // T is explicitly specified as int
}
```

*Kenneth J. Roche*
*University of Washington*

```cpp
// variable template
template <class T>
constexpr T pi = T(3.14159265358979323846264433L);
```

```
Pi: 3
Pi: 3.14159
Pi: 3.14159265358979
```

```cpp
// variable template
std::cout << "Pi: " << pi<int> << std::endl;
std::cout << "Pi: " << pi<float> << std::endl;
std::cout << "Pi: " << std::setprecision(15) << pi<double> << std::endl;
```

*templates - class templates (more to say …)*

- *variable template*
- *non-type parameters*
- *class and function template specialization*
- *variadic templates*
- *fold expressions (…)*
- *template lambdas*

*Default types, etc.*

```cpp
// non-type parameters
template <class T = int , const int n = 3>
class myStorage
{
public:
    T store[n];
};
myStorage<> default_box;
myStorage<double,10> dbox;
```

*Kenneth J. Roche*
*University of Washington*

## *class constructors*

- *special member function that is called automatically when an object of a class is created*
- *initializes the data members of the class*
- *has the same name as the class*
- *does not have a return type*
- *is declared in the public section of the class*
- *can be overloaded to take different sets of parameters*

## *class destructors*

- *frees allocated resources*
- *called automatically before an object destroyed*
  - *called as object goes out of scope*
  - *or when explicitly deleted for objects allocated with new*
- *has the same name as the class preceded by a ~*
- *takes no arguments*
- *does not have a return type*
- *classes have only one destructor*

- *to be accessible from outside the class, the constructor is declared in public*
- *same for destructor*

```
class rectangle
{
public:
    int x, y;
    int area();   // method declaration
    rectangle();  // constructor
    ~rectangle(); // destructor
};
rectangle::rectangle() { std::cout << "rectangle constructed" << std::endl; }
rectangle::~rectangle() { std::cout << "rectangle destructed" << std::endl; }
int rectangle::area() { return x * y; }
```

```
rectangle r1;
r1.x = 3;
r1.y = 4;
std::cout << "area: " << r1.area() << std::endl;
```

```
rectangle constructed
area: 12
rectangle destructed
```

```
rectangle constructed
area: 12
rectangle overload constructor
area r3: 20
rectangle destructed
rectangle destructed
```

```cpp
int main()
{

    rectangle r1;
    r1.x = 3;
    r1.y = 4;
    std::cout << "area: " << r1.area() << std::endl;

    rectangle r3(4, 5);
    std::cout << "area r3: " << r3.area() << std::endl;

    return 0; // fast return
}
```

```cpp
class rectangle
{
public:
    int x, y;
    int area();           // method declaration
    rectangle();          // constructor
    rectangle(int, int); // overload constructor
    ~rectangle();         // destructor
};
rectangle::rectangle() { std::cout << "rectangle constructed" << std::endl; }
rectangle::rectangle(int a, int b)
{
    x = a; y = b;
    std::cout << "rectangle overload constructor" << std::endl;
}
rectangle::~rectangle() { std::cout << "rectangle destructed" << std::endl; }
int rectangle::area() { return x * y; }
```

- *constructor overloading and default initialization*

*Kenneth J. Roche*
*University of Washington*

```
rectangle constructed
area: 12
rectangle overload constructor
area r3: 20
rectangle destructed
rectangle destructed
```

```cpp
int main()
{

    rectangle r1;
    r1.x = 3;
    r1.y = 4;
    std::cout << "area: " << r1.area() << std::endl;

    rectangle r3(4, 5);
    std::cout << "area r3: " << r3.area() << std::endl;

    return 0; // fast return
}
```

```cpp
class rectangle
{
public:
    int x, y;
    int area();           // method declaration
    rectangle();          // constructor
    rectangle(int, int); // overload constructor
    ~rectangle();         // destructor
};
rectangle::rectangle() { std::cout << "rectangle constructed" << std::endl; }
rectangle::rectangle(int a, int b) : x(a),y(b)
{std::cout << "rectangle overload constructor" << std::endl;}
rectangle::~rectangle() { std::cout << "rectangle destructed" << std::endl; }
int rectangle::area() { return x * y; }
```

- *constructor overloading and default initialization*
- *constructor initializer list*

*Kenneth J. Roche*
*University of Washington*
*AMATH 483 / 583 High Performance Scientific Computing*
*Spring Quarter 2025*

*class copy constructor*

- *compiler also provides a default copy constructor*

- *is called each time a copy of a class object is made*
  - *passing an object by value to functions*
  - *returning an object by value from a function*

- *only take a single parameter: a reference to an object of the class*

- *default copy constructor copies each member variable from the passed object to the member variables of the new object*

# *class copy constructor -shallow copy (pointers in both objects end up pointing to the same memory)*

```
class rectangle
{
public:
    int x, y;
    int area();           // method declaration
    rectangle();          // constructor
    rectangle(int, int); // overload constructor
    rectangle(const rectangle&);  // copy constructor
    ~rectangle();         // destructor
};
rectangle::rectangle() { std::cout << "rectangle constructed" << std::endl; }

rectangle::rectangle(int a, int b) : x(a),y(b)
{std::cout << "rectangle overload constructor" << std::endl;}

rectangle::rectangle(const rectangle& other) : x(other.x), y(other.y)
{std::cout << "rectangle copy constructor" << std::endl;}

rectangle::~rectangle() { std::cout << "rectangle destructed" << std::endl; }

int rectangle::area() { return x * y; }
```

*Kenneth J. Roche*
*University of Washington*

*class copy constructor -shallow copy (pointers in both objects end up pointing to the same memory) (out of scope??)*

```
class rectangle
{
public:
    int x, y;
    int area();          // method declaration
    rectangle();         // constructor
    rectangle(int, int); // overload constructor
    rectangle(const rectangle&);  // copy constructor
    ~rectangle();        // destructor
};
rectangle::rectangle() { std::cout << "rectangle constructed" << std::endl; }

rectangle::rectangle(int a, int b) : x(a),y(b)
{std::cout << "rectangle overload constructor" << std::endl;}

rectangle::rectangle(const rectangle& other) : x(other.x), y(other.y)
{std::cout << "rectangle copy constructor" << std::endl;}

rectangle::~rectangle() { std::cout << "rectangle destructed" << std::endl; }

int rectangle::area() { return x * y; }
```

```
rectangle r1;
r1.x = 3;
r1.y = 4;
std::cout << "area: " << r1.area() << std::endl;

rectangle r3(4, 5);
std::cout << "area r3: " << r3.area() << std::endl;

rectangle r4 = rectangle(r3); //shallow copy
std::cout << "area r4: " << r4.area() << std::endl;

return 0; // fast return
```

```
rectangle constructed
area: 12
rectangle overload constructor
area r3: 20
rectangle copy constructor
area r4: 20
rectangle destructed
rectangle destructed
rectangle destructed
```

*Kenneth J. Roche*
*University of Washington*

```cpp
int main()
{
    rectangle r1;
    r1.x = 3;
    r1.y = 4;
    std::cout << "area: " << r1.area() << std::endl;

    rectangle r3(4, 5);
    std::cout << "area r3: " << r3.area() << std::endl;

    rectangle r4 = rectangle(r3); // shallow copy
    std::cout << "area r4: " << r4.area() << std::endl;

    rectangle *r5 = new rectangle(r1);
    std::cout << "area r5: " << r5->area() << std::endl;

    (*r5).x = 6; (*r5).y = 7;
    std::cout << "area r5: " << r5->area() << std::endl;
    std::cout << "area r1: " << r1.area() << std::endl;

    delete r5; //clean up the heap
    return 0; // fast return
}
```

*class copy constructor -deep copy (new memory)*

```
rectangle constructed
area: 12
rectangle overload constructor
area r3: 20
rectangle copy constructor
area r4: 20
rectangle copy constructor
area r5: 12
area r5: 42
area r1: 12
rectangle destructed
rectangle destructed
rectangle destructed
rectangle destructed
```

*poor man's matrix class*

- *constructor*
- *destructor*
- *copy constructor*
- *class methods for accessing private elements*
- *operator overloading for class members*

*poor man's matrix class*

- *constructor*
- *destructor*
- *copy constructor*
- *class methods for accessing private elements*
- *operator overloading for class members*

```cpp
// simple matrix class
class Matrix
{
public:
    Matrix(int, int);            // constructor
    ~Matrix();                   // destructor
    Matrix(const Matrix &other); // copy constructor

    // accessor methods - class functions that can access private foo
    int getRows() const { return rows_; }
    int getCols() const { return cols_; }
    double get_ij(int i, int j) const { return matrix_[i][j]; }
    void set_ij(int i, int j, double value) { matrix_[i][j] = value; }
    void print() const;

    // alternate reference notation ... A[i][j]
    // element access operators
    std::vector<double> &operator[](int i) { return matrix_[i]; }
    const std::vector<double> &operator[](int i) const { return matrix_[i]; }

    Matrix operator*(const Matrix &other) const; // matrix multiply
    Matrix operator+(const Matrix &other) const; // matrix addition
    Matrix operator*(double scalar) const;       // scale matrix
    Matrix operator-(const Matrix &other) const; // matrix subtraction is redundant

private:
    std::vector<std::vector<double>> matrix_;
    int rows_;
    int cols_;
};
```

*poor man's matrix class*

- *constructor*
- *class methods for accessing private elements*

```
Matrix m(3, 4); // creates a 3×4 matrix
```

```
Matrix::Matrix(int rows, int cols)
    : matrix_(rows, std::vector<double>(cols)), rows_(rows), cols_(cols)
{ // matrix constructor
}
```

```
private:
    std::vector<std::vector<double>> matrix_;
    int rows_;
    int cols_;
};
```

```cpp
Matrix::~Matrix()
{
    // deallocate the memory used by the vector of vectors
    matrix_.clear();
    std::cout << "Matrix destructed" << std::endl;
}

Matrix::Matrix(int rows, int cols)
    : matrix_(rows, std::vector<double>(cols)), rows_(rows), cols_(cols)
{ // matrix constructor
}

Matrix::Matrix(const Matrix &other)
    : rows_(other.rows_), cols_(other.cols_), matrix_(other.matrix_)
{ // copy constructor
}
```

*poor man's matrix class*

- *class methods for accessing private elements*

```
private:
    std::vector<std::vector<double>> matrix_;
    int rows_;
    int cols_;
};
```

```
// accessor methods - class functions that can access private foo
int getRows() const { return rows_; }
int getCols() const { return cols_; }
double get_ij(int i, int j) const { return matrix_[i][j]; }
void set_ij(int i, int j, double value) { matrix_[i][j] = value; }
void print() const;
// alternate reference notation ... A[i][j]
// element access operators
std::vector<double> &operator[](int i) { return matrix_[i]; }
const std::vector<double> &operator[](int i) const { return matrix_[i]; }
```

The first operator[] returns a reference to the vector of double values at row i, which can then be indexed with j to retrieve the matrix element at position (i, j).

*poor man's matrix class*

- *class methods for accessing private elements*

```
// accessor methods - class functions that can access private foo
int getRows() const { return rows_; }
int getCols() const { return cols_; }
double get_ij(int i, int j) const { return matrix_[i][j]; }
void set_ij(int i, int j, double value) { matrix_[i][j] = value; }
void print() const;
// alternate reference notation ... A[i][j]
// element access operators
std::vector<double> &operator[](int i) { return matrix_[i]; }
const std::vector<double> &operator[](int i) const { return matrix_[i]; }
```

```
void Matrix::print() const
{
    for (int i = 0; i < rows_; i++)
    {
        for (int j = 0; j < cols_; j++)
        {
            std::cout << matrix_[i][j] << " ";
        }
        std::cout << std::endl;
    }
}
```

*poor man's matrix class*

- *operator overloading for class members*

```
Matrix operator*(const Matrix &other) const; // matrix multiply
```

`operator*`: This is the name of the operator being overloaded. In this case, it is the multiplication operator `*`.

`const Matrix &other`: This is the argument to the operator overload. It is a constant reference to another `Matrix` object that will be multiplied with the current object. The `const` qualifier ensures that the argument cannot be modified within the function.

`const`: This keyword specifies that the function does not modify the state of the `Matrix` object it is called on. It is part of the function signature and allows the function to be called on `const` objects of the `Matrix` class.

`Matrix`: This is the return type of the function. In this case, the `operator*` overload returns a new `Matrix` object that represents the result of the matrix multiplication operation.

```cpp
Matrix operator*(const Matrix &other) const; // matrix multiply
```

*poor man's matrix class*

- *operator overloading for class members*

```cpp
Matrix Matrix::operator*(const Matrix &other) const
{
    if (cols_ != other.rows_)
    {
        throw std::invalid_argument("Matrices are not compatible for multiplication");
    }
    Matrix result(rows_, other.cols_);
    for (int i = 0; i < rows_; i++)
    {
        for (int j = 0; j < other.cols_; j++)
        {
            double sum = 0;
            for (int k = 0; k < cols_; k++)
            {
                sum += matrix_[i][k] * other.matrix_[k][j];
            }
            result.set_ij(i, j, sum);
        }
    }
    return result;
}
```

```cpp
// example of simple matrix class with operator overloading
// Initialize matrices
Matrix A(2, 2); // constructor invoked ...
A.set_ij(0, 0, 1.0);
A.set_ij(0, 1, 2.0);
A.set_ij(1, 0, 3.0);
A.set_ij(1, 1, 4.0);

Matrix F(A); // copy A using the copy constructor

// check the alternate access notation
double v = A[0][1];
std::cout << "w = A[0][1] = " << v << std::endl;

Matrix B(2, 2);
B.set_ij(0, 0, 5.0);
B.set_ij(0, 1, 6.0);
B.set_ij(1, 0, 7.0);
B.set_ij(1, 1, 8.0);

// Matrix multiplication
Matrix C = A * B;
std::cout << "Matrix C = A * B:" << std::endl;
C.print();

// Matrix addition
Matrix D = A + B;
std::cout << "Matrix D = A + B:" << std::endl;
D.print();

// Scalar multiplication
Matrix E = 2.0 * A;
std::cout << "Matrix E = 2. * A:" << std::endl;
E.print();
```

```
w = A[0][1] = 2
Matrix C = A * B:
19 22
43 50
Matrix D = A + B:
6 8
10 12
Matrix E = 2 * A:
2 4
6 8
Matrix destructed
Matrix destructed
Matrix destructed
Matrix destructed
Matrix destructed
Matrix destructed
Matrix destructed
```

*inheritance*

- *allows a class to acquire the members of another class*

```
//inheritance ...
class square : public rectangle{};
// rectangle is the base class of square
// square is derived from rectangle
//square does not define any new member
//functions or variables,
//but it can use all of the member functions
//and variables of rectangle
```

```
int main()
{
    rectangle r1;
    r1.x = 3;
    r1.y = 4;
    std::cout << "area: " << r1.area() << std::endl;

    square s1;
    s1.x = 4; s1.y=5;
    std::cout << "area s1: " << s1.area() << std::endl;
}
```

```
rectangle constructed
area: 12
rectangle constructed
area s1: 20
rectangle destructed
rectangle destructed
```

*inheritance*

- *objects can be upcast to their base class*
- *assign derived object to reference of base class*
- *assign derived object to a pointer of the base class*

```
//inheritance ...
class square : public rectangle{};
// rectangle is the base class of square
// square is derived from rectangle
//square does not define any new member
//functions or variables,
//but it can use all of the member functions
//and variables of rectangle
```

```
//allows accessing only the public members of rectangle
rectangle& r6 = s1; //reference upcast
rectangle* r7 = &s1; //pointer upcast
```

*inheritance*

- *an upcast object can be downcast to their base class*
- *explicit casting*

```
// multiple inheritance
class people {};
class employee {};
class professor : public people, public employee {};
```

```
//allows accessing only the public members of rectangle
rectangle& r6 = s1; //reference upcast
rectangle* r7 = &s1; //pointer upcast
```

```
//downcast by explicit cast
square& sq1 = static_cast<square&>(r6);
square* sq2 = static_cast<square*>(r7);
```

*inheritance*

- *multiple inheritance*

```
// multiple inheritance
class people {};
class employee {};
class professor : public people, public employee {};
```

*inheritance overriding*

- *a method in a derived class can redefine a method in the base class*

*TO BE CONTINUED ….*

*… comments on HW2 ….*

*End Lecture 7*

*Kenneth J. Roche*
*University of Washington*
*AMATH 483 / 583 High Performance Scientific Computing*
*Spring Quarter 2025*