lecture 10

- *Poor Man's matrix class*
    - *defining class operators for +,-,*,/*
- *Inheritance -intro*
- *Strassen's matrix multiply algorithm*

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

poor man's matrix class

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

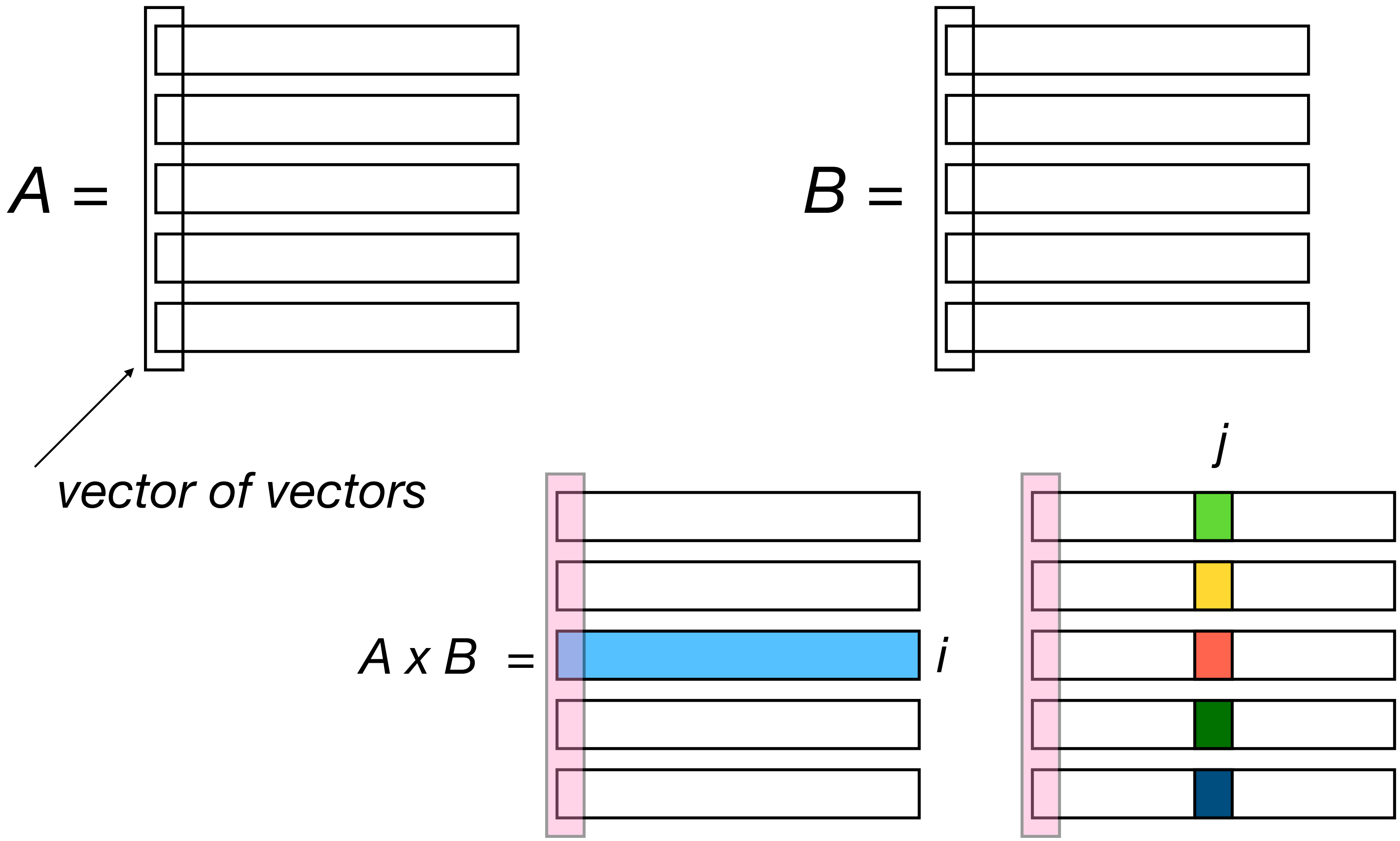
poor man's matrix class

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

```
// 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_;
};
```

Kenneth J. Roche
University of Washington

matrix foo revisited ….



poor man's matrix class

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

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

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

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

Kenneth J. Roche
University of Washington

```
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
}
```

```
std::vector<std::vector<double>>().swap(matrix);
```

Creates a temporary, **empty** `std::vector<std::vector<double>>`.

Swaps it with your existing `matrix`.

Kenneth J. Roche
University of Washington

poor man's matrix class

- *class methods for accessing private elements*

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

```cpp
// 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).

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

poor man’s matrix class

- *class methods for accessing private elements*

```cpp
// 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]; }
```

```cpp
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.

poor man's matrix class

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

- *operator overloading for class members*

```
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;
}
```

`A` calls the `operator*` method.

`other` becomes `B`.

A new `Matrix` `C` is created inside the method and returned — `A` and `B` are not modified.

```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
```

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

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
```

Kenneth J. Roche
University of Washington

inheritance

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

```cpp
#ifndef AUTOMOBILE_HPP
#define AUTOMOBILE_HPP

#include <string>
#include <iostream>

class Automobile {
protected:
    std::string brand;
    int year;

public:
    Automobile(const std::string& brand, int year)
        : brand(brand), year(year) {}

    void showDetails() const {
        std::cout << "Brand: " << brand << ", Year: " << year << std::endl;
    }
};

#endif // AUTOMOBILE_HPP
```

```cpp
#ifndef CAR_HPP
#define CAR_HPP

#include "automobile.hpp"

class Car : public Automobile {
private:
    int num_doors;

public:
    Car(const std::string& brand, int year, int doors)
        : Automobile(brand, year), num_doors(doors) {}

    void showCar() const {
        showDetails();
        std::cout << "Type: Car, Doors: " << num_doors << std::endl;
    }
};
```

```cpp
#ifndef TRUCK_HPP
#define TRUCK_HPP

#include "automobile.hpp"

class Truck : public Automobile {
private:
    double payload_capacity;

public:
    Truck(const std::string& brand, int year, double capacity)
        : Automobile(brand, year), payload_capacity(capacity) {}

    void showTruck() const {
        showDetails();
        std::cout << "Type: Truck, Payload Capacity: " << payload_capacity << " tons" << std::endl;
    }
};
```

```cpp
#include "car.hpp"
#include "truck.hpp"

int main() {
    Car myCar("Toyota", 2022, 4);
    Truck myTruck("Ford", 2020, 2.5);

    myCar.showCar();
    std::cout << std::endl;
    myTruck.showTruck();

    return 0;
}
```

```
bash-3.2$ vi automobile.hpp
bash-3.2$ vi car.hpp
bash-3.2$ vi truck.hpp
bash-3.2$ vi inherit1.cpp
bash-3.2$ g++ -std=c++17 -o xinherit1 -I./ inherit1.cpp
bash-3.2$ ./xinherit1
Brand: Toyota, Year: 2022
Type: Car, Doors: 4

Brand: Ford, Year: 2020
Type: Truck, Payload Capacity: 2.5 tons
bash-3.2$
```

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

inheritance

- *allows a class to acquire the members of another class*
- *virtual functions + polymorphism: lets us call the correct method on derived classes via a base class pointer or reference*

Concept	Description
`virtual`	Declares a method meant to be overridden by derived classes. Enables polymorphism.
`override`	Ensures a derived method is actually overriding a virtual base method.

Kenneth J. Roche
University of Washington

```cpp
#ifndef AUTOMOBILE_HPP
#define AUTOMOBILE_HPP

#include <string>
#include <iostream>

class Automobile {
protected:
    std::string brand;
    int year;

public:
    Automobile(const std::string& brand, int year)
        : brand(brand), year(year) {}

    // Virtual function to allow overriding
    virtual void showInfo() const {
        std::cout << "Automobile - Brand: " << brand << ", Year: " << year << std::endl;
    }

    // Always good practice: a virtual destructor for base classes
    virtual ~Automobile() = default;
};

#endif // AUTOMOBILE_HPP
```

```cpp
#ifndef CAR_HPP
#define CAR_HPP

#include "automobile.hpp"

class Car : public Automobile {
private:
    int num_doors;

public:
    Car(const std::string& brand, int year, int doors)
        : Automobile(brand, year), num_doors(doors) {}

    void showInfo() const override {
        std::cout << "Car - Brand: " << brand << ", Year: " << year
                  << ", Doors: " << num_doors << std::endl;
    }
};
#endif // CAR_HPP
```

```cpp
#ifndef TRUCK_HPP
#define TRUCK_HPP

#include "automobile.hpp"

class Truck : public Automobile {
private:
    double payload_capacity;

public:
    Truck(const std::string& brand, int year, double capacity)
        : Automobile(brand, year), payload_capacity(capacity) {}

    void showInfo() const override {
        std::cout << "Truck - Brand: " << brand << ", Year: " << year
                  << ", Payload: " << payload_capacity << " tons" << std::endl;
    }
};
#endif // TRUCK_HPP
```

```cpp
#include "car.hpp"
#include "truck.hpp"
#include <vector>

int main() {
    // Create instances
    Car car("Honda", 2023, 4);
    Truck truck("Volvo", 2019, 7.5);

    // Store as pointers to base class
    std::vector<Automobile*> garage;
    garage.push_back(&car);
    garage.push_back(&truck);

    // Call polymorphic function
    for (const auto* vehicle : garage) {
        vehicle->showInfo();  // Dynamically calls Car/Truck version
    }

    return 0;
}
```

The function `showInfo()` is **virtual** in the base class, and **overridden** in derived classes.

Even though we use `Automobile*` in the loop, **the correct derived class version** is invoked.

This is **runtime polymorphism** via **dynamic dispatch**.

```
bash-3.2$ g++ -std=c++17 -o xinherit1 -I./ inherit1.cpp
bash-3.2$ ./xinherit1
Car - Brand: Honda, Year: 2023, Doors: 4
Truck - Brand: Volvo, Year: 2019, Payload: 7.5 tons
bash-3.2$
```

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 safely always*
- *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 -upcast

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

🔼 Upcasting

Definition: Converting a derived class pointer or reference to a base class type.

✅ **Always safe** — no cast operator required.

cpp Copy Edit

```cpp
class Vehicle {
public:
    virtual void drive() {}
};

class Car : public Vehicle {
public:
    void drive() override {}
};

Car myCar;
Vehicle* vPtr = &myCar; // ✅ Upcasting — safe and implicit
```

inheritance -downcast

- *use with care*

🔽 Downcasting

Definition: Converting a base class pointer/reference to a derived class type.

⚠️ **Potentially unsafe** — must ensure the base class actually points to the correct derived type.

```cpp
Vehicle* vPtr = new Car();

Car* cPtr = dynamic_cast<Car*>(vPtr);  // ✅ Safe if vPtr really points to a Car
```

- Use `dynamic_cast` (requires at least one **virtual** function in the base class).
- Returns `nullptr` if the cast is invalid (for pointers).
- If using references, `dynamic_cast<Car&>(vRef)` throws `std::bad_cast` on failure.

inheritance

- *multiple inheritance*

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

matrix multiply revisited …. divide and conquer ala Strassen

- *2 x 2 multiplication can be achieve with 7 multiplies - not 8*
- *tradeoffs*
  - *increase in storage*
  - *number of additions goes from 4 to 18*

$$\begin{bmatrix} C(1,1) & C(1,2) \\ C(2,1) & C(2,2) \end{bmatrix} = \begin{bmatrix} A(1,1) & A(1,2) \\ A(2,1) & A(2,2) \end{bmatrix} \begin{bmatrix} B(1,1) & B(1,2) \\ B(2,1) & B(2,2) \end{bmatrix}$$

$$C = A\,B$$

$$P1 = (A(1,1)+A(2,2))*(B(1,1)+B(2,2))$$
$$P2 = (A(2,1)+A(2,2))*B(1,1)$$
$$P3 = A(1,1)*(B(1,2)-B(2,2))$$
$$P4 = A(2,2)*(B(2,1)-B(1,1))$$
$$P5 = (A(1,1)+A(1,2))*B(2,2)$$
$$P6 = (A(2,1)-A(1,1))*(B(1,1)+B(1,2))$$
$$P7 = (A(1,2)-A(2,2))*(B(2,1)+B(2,2))$$

$$C(1,1) = P1 + P4 - P5 + P7 \qquad C(1,2) = P3 + P5$$
$$C(2,1) = P2 + P4 \qquad C(2,2) = P1 + P3 - P2 + P6$$

matrix multiply revisited …. divide and conquer ala Strassen

- *2 x 2 multiplication can be achieve with 7 multiplies - not 8*
- *tradeoffs*
  - *increase in storage*
  - *number of additions goes from 4 to 18*
- *complexity*
  - *O(4.7 n^2.81) vs O(2 n^3)*
    - *n=1000: 2n^3=2e9 ; 4.7n^2.81~1.27e9*

- *n x n multiplication, n even*
- *partition matrices into (n/2) x (n/2) blocks*
  - *multiplies ~ 2(n/2)^3*
  - *adds ~ (n/2)^2*
- *complexity*
  - *7 x 2(n/2)^3 + 18 x (n/2)^2 = (7/4)n^3 + (9/2)n^2*
  - *for n > 18, Strassen has less complex operation count*

End Lecture 10

Kenneth J. Roche
University of Washington