lectures 8 (2 of 2 slide decks -maybe more needs to be covered)

- *C++ at a Glance - introduction to …*
  - *default template clarification*
  - *more about pointers*
    - *new / delete*
    - *smart pointers (unique, shared)*
  - *class constructors, destructors, copy constructor*
  - *introduction to the poor man's matrix class*
    - *defining class operators for +,-,*,/*

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

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

set defaults

```
#include <iostream>

// Template definition
template <class T = int, const int n = 3>
class myStorage
{
public:
    T store[n];
};

// Function to print elements of myStorage
template <class T, int n>
void printStorage(const myStorage<T, n> &box)
{
    for (int i = 0; i < n; ++i)
    {
        std::cout << box.store[i] << ' ';
    }
    std::cout << std::endl;
}
```

```
int main()
{
    // Using default template parameters: T = int, n = 3
    myStorage<> default_box;
    default_box.store[0] = 1;
    default_box.store[1] = 2;
    default_box.store[2] = 3;

    // Using specified template parameters: T = double, n = 10
    myStorage<double, 10> box;
    for (int i = 0; i < 10; ++i)
    {
        box.store[i] = i * 1.1; // Assigning some values
    }

    // Accessing elements in main
    std::cout << "default_box.store[1] = " << default_box.store[1] << std::endl;
    std::cout << "box.store[5] = " << box.store[5] << std::endl;

    // Using the printStorage function
    std::cout << "Contents of default_box: ";
    printStorage(default_box);

    std::cout << "Contents of box: ";
    printStorage(box);

    return 0;
}
```

clarification - was a little vague last lecture on this slide

Kenneth J. Roche
University of Washington

```cpp
int main()
{
    // Using default template parameters: T = int, n = 3
    myStorage<> default_box;
    default_box.store[0] = 1;
    default_box.store[1] = 2;
    default_box.store[2] = 3;

    // Using specified template parameters: T = double, n = 10
    myStorage<double, 10> box;
    for (int i = 0; i < 10; ++i)
    {
        box.store[i] = i * 1.1; // Assigning some values
    }

    // Accessing elements in main
    std::cout << "default_box.store[1] = " << default_box.store[1] << std::endl;
    std::cout << "box.store[5] = " << box.store[5] << std::endl;

    // Using the printStorage function
    std::cout << "Contents of default_box: ";
    printStorage(default_box);

    std::cout << "Contents of box: ";
    printStorage(box);

    return 0;
}
```

```cpp
#include <iostream>

// Template definition
template <class T = int, const int n = 3>
class myStorage
{
public:
    T store[n];
};

// Function to print elements of myStorage
template <class T, int n>
void printStorage(const myStorage<T, n> &box)
{
    for (int i = 0; i < n; ++i)
    {
        std::cout << box.store[i] << ' ';
    }
    std::cout << std::endl;
}
```

```
bash-3.2$ g++ -std=c++17 -c mystorage.cpp ; g++ -o xmystorage mystorage.o
bash-3.2$ ./xmystorage
default_box.store[1] = 2
box.store[5] = 5.5
Contents of default_box: 1 2 3
Contents of box: 0 1.1 2.2 3.3 4.4 5.5 6.6 7.7 8.8 9.9
bash-3.2$
```

clarification - was a little vague last lecture on this slide

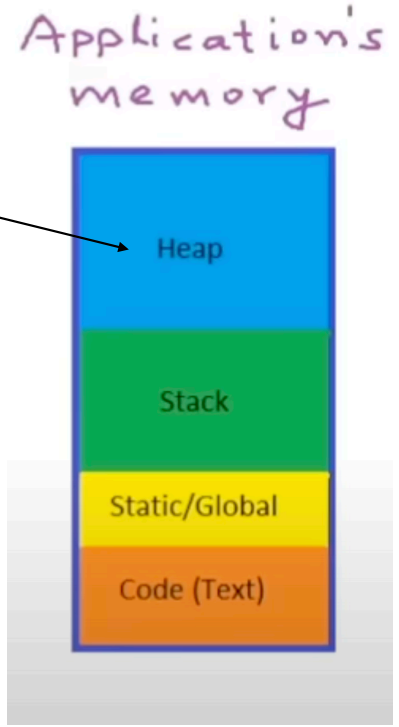
Kenneth J. Roche
University of Washington

more about pointers

- *new / delete*

```cpp
#include <iostream>
#include <vector>

int main() {
    // Dynamically allocate a vector of integers
    std::vector<int>* vec = new std::vector<int>{1, 2, 3, 4, 5};

    // Access and modify elements
    (*vec)[2] = 10;

    // Print elements
    std::cout << "Vector elements: ";
    for (int val : *vec) {
        std::cout << val << " ";
    }
    std::cout << std::endl;

    // Dynamically allocate a vector of vectors (2D vector)
    std::vector<std::vector<int>>* matrix = new std::vector<std::vector<int>>(3, std::vector<int>(4, 0));

    // Modify elements
    (*matrix)[1][2] = 7;

    // Print matrix
    std::cout << "Matrix elements:" << std::endl;
    for (const auto& row : *matrix) {
        for (int val : row) {
            std::cout << val << " ";
        }
        std::cout << std::endl;
    }

    // Deallocate memory
    delete vec;
    delete matrix;

    return 0;
}
```

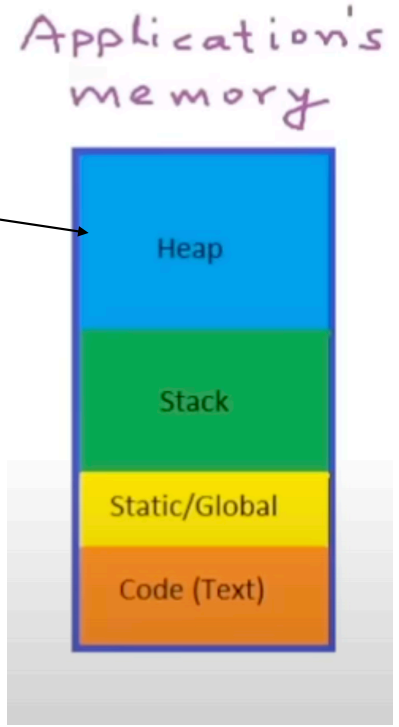
```
bash-3.2$ g++ -std=c++17 -o xdynmem1 dyn_mem1.cpp
bash-3.2$ ./xdynmem1
Vector elements: 1 2 10 4 5
Matrix elements:
0 0 0 0
0 0 7 0
0 0 0 0
bash-3.2$
```



Kenneth J. Roche
University of Washington

```cpp
#include <iostream>
#include <vector>

// Function that returns a dynamically allocated vector
std::vector<int>* createVector(int size, int initialValue) {
    return new std::vector<int>(size, initialValue);
}

int main() {
    // Create a vector of size 5, initialized with 42
    std::vector<int>* myVec = createVector(5, 42);

    // Print elements
    std::cout << "Created vector: ";
    for (int val : *myVec) {
        std::cout << val << " ";
    }
    std::cout << std::endl;

    // Deallocate memory
    delete myVec;

    return 0;
}
```

- *more about pointers*
  - *new / delete*

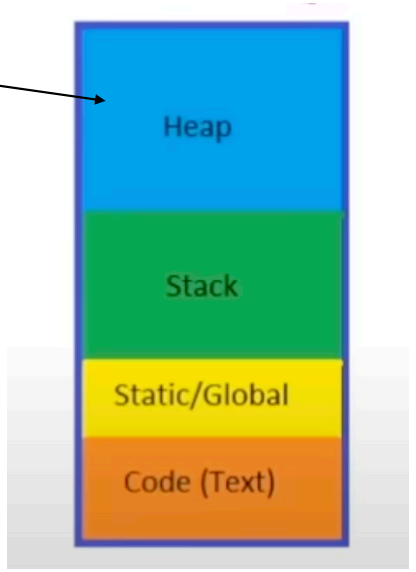



```
bash-3.2$ g++ -std=c++17 -o xdynmemfnc dyn_mem_fnc.cpp
bash-3.2$ ./xdynmemfnc
Created vector: 42 42 42 42 42
bash-3.2$
```

Kenneth J. Roche
University of Washington

- *C++ smart pointers*
  - *A unique_ptr owns a resource exclusively. When the unique_ptr goes out of scope, it deletes the resource*

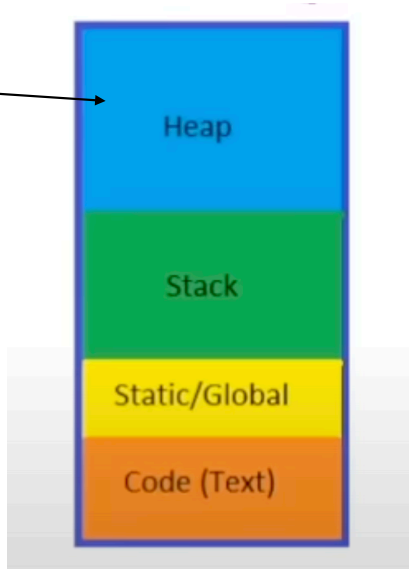
```cpp
#include <iostream>
#include <memory>
#include <vector>

int main()
{
    // Create a unique_ptr to a vector
    std::unique_ptr<std::vector<int>> vecPtr = std::make_unique<std::vector<int>>(std::initializer_list<int>{1, 2, 3});

    // Access and modify elements
    (*vecPtr)[1] = 20;

    // Print elements
    std::cout << "Unique_ptr vector: ";
    for (int val : *vecPtr)
    {
        std::cout << val << " ";
    }
    std::cout << std::endl;

    // No need to delete; memory is automatically managed
    return 0;
}
```



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

- *C++ smart pointers*
  - ***unique_ptr*** *owns a resource exclusively*
    - *one* ***unique_ptr*** *can own a particular resource at a time*
    - *copying a* ***unique_ptr*** *is not allowed*
    - *ownership can be transferred using* ***std::move***

```cpp
#include <iostream>
#include <memory>

int main()
{
    std::unique_ptr<int> ptr1 = std::make_unique<int>(10);
    // std::unique_ptr<int> ptr2 = ptr1; // Error: copy constructor is deleted

    std::unique_ptr<int> ptr2 = std::move(ptr1); // Ownership transferred to ptr2

    if (!ptr1)
    {
        std::cout << "ptr1 is now null after move." << std::endl;
    }

    std::cout << "ptr2 owns the resource with value: " << *ptr2 << std::endl;

    return 0;
}
```



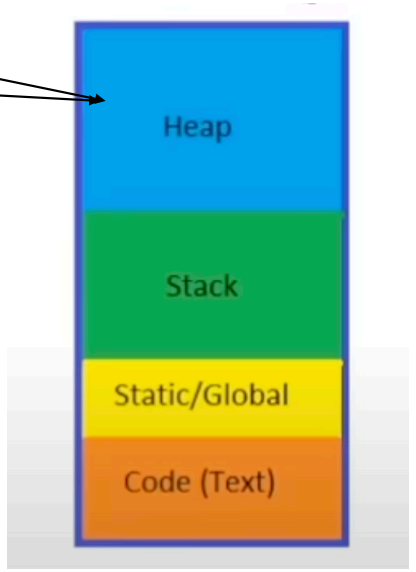

```
bash-3.2$ g++ -std=c++17 -o xdynmemunq2 dyn_mem_smart_unq2.cpp
bash-3.2$ ./xdynmemunq2
ptr1 is now null after move.
ptr2 owns the resource with value: 10
bash-3.2$
```

Kenneth J. Roche
University of Washington

- *C++ smart pointers*
  - ***shared_ptr*** *allows multiple* ***shared_ptr*** *instances on a resource*
    - *a reference count is used to determine when to deallocate the resource*

```cpp
#include <iostream>
#include <memory>

int main()
{
    std::shared_ptr<int> ptr1 = std::make_shared<int>(20);
    std::cout << "ptr1 use_count: " << ptr1.use_count() << std::endl;

    std::shared_ptr<int> ptr2 = ptr1; // Shared ownership
    std::cout << "After ptr2 = ptr1;" << std::endl;
    std::cout << "ptr1 use_count: " << ptr1.use_count() << std::endl;
    std::cout << "ptr2 use_count: " << ptr2.use_count() << std::endl;

    ptr1.reset(); // Releases ownership from ptr1
    std::cout << "After ptr1.reset();" << std::endl;
    std::cout << "ptr1 use_count: " << ptr1.use_count() << std::endl;
    std::cout << "ptr2 use_count: " << ptr2.use_count() << std::endl;

    return 0;
}
```

```
bash-3.2$ g++ -std=c++17 -o xdynmemshrd dyn_mem_smart_shrd.cpp
bash-3.2$ ./xdynmemshrd
ptr1 use_count: 1
After ptr2 = ptr1;
ptr1 use_count: 2
ptr2 use_count: 2
After ptr1.reset();
ptr1 use_count: 0
ptr2 use_count: 1
bash-3.2$
```



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
```

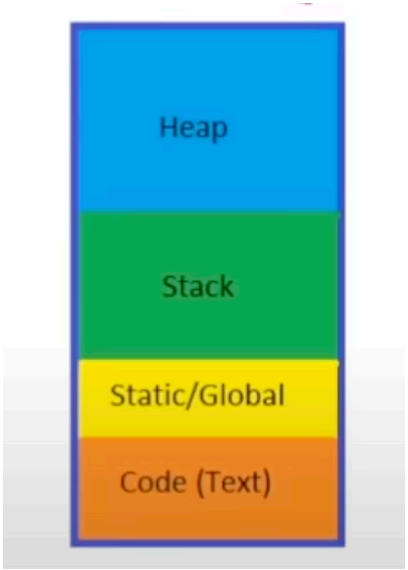
```
rectangle constructed
area: 12
rectangle overload constructor
area r3: 20
rectangle copy constructor
area r4: 20
rectangle destructed
rectangle destructed
rectangle destructed
```

```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
```

- *C++ memory management* ***Rule of Three:***
  - ***Destructor:*** *to release the allocated resources*
  - ***Copy Constructor:*** *to create a new object as a copy of an existing object*
  - ***Copy Assignment Operator:*** *to assign the contents of one existing object to another existing object*



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_;
};
```

End Lecture 8