Rule of 5
Templates
Topics

1. Rule of 5 (The Big 5)
2. Templates
Rule of Three

The **rule of three** is a rule of thumb in C++ (prior to C++11) that claims that if a class defines any of the following then it should probably explicitly define all three:

- destructor
- copy constructor
- copy assignment operator
Rule of Five (The Big 5)

The **Rule of Five** is a rule of thumb in C++ (prior to C++11) that claims that if a class defines any of the following then it should probably explicitly define all five:

- destructor
- copy constructor
- copy assignment operator
- **Move constructor**
- **Move assignment operator**

“Because the presence of a user-defined destructor, copy-constructor, or copy-assignment operator prevents implicit definition of the move constructor and the move assignment operator, any class for which move semantics are desirable, has to declare all five special member functions”
Rule of Five (The Big 5)

“Because the presence of a user-defined destructor, copy-constructor, or copy-assignment operator prevents implicit definition of the move constructor and the move assignment operator, any class for which move semantics are desirable, has to declare all five special member functions”

Array2D has a user-defined destructor. Why?
The Rule of Zero

If possible, WRITE NO DEFAULT OPERATIONS.
move constructor

Since we previously wrote the destructor, copy constructor and copy assignment operator for Array2D, we’ll look now at move constructor and the move assignment operator.

The move constructor works in a manner similar to the copy constructor. Recall that a copy constructor was user-defined when Array2D dynamically allocated memory.
Array2D: copy constructor

Array2D::Array2D(const Array2D& arr) {
    name = arr.name;
    rows = arr.rows;
    columns = arr.columns;
    count = arr.count;
    myInts = new int[count];  // allocates memory
    std::memmove(myInts, arr.myInts, count*sizeof(int));  //moves contents
}

Array2D: move constructor

Array2D::Array2D(const Array2D& arr) {
    name = arr.name;
    rows = arr.rows;
    columns = arr.columns;
    count = arr.count;
    myInts = arr.myInts;  // copies pointer
    arr.myInts = nullptr; // severs ties with data
}
move constructor

This works in a manner similar to the copy constructor. Recall that a copy constructor is user-defined when an object dynamically allocated memory as we have shown with Array2D.

1. The move constructor is different in that it “steals” the data from the source by
   a. copying the data area pointer(s) that point to the data area(s) and
   b. setting the pointer(s) in the original object to “nullptr”.
2. No memory is moved or copied.
3. The original object loses access to the data area.
move assignment operator

This performs a move in response to an assignment to an rvalue*. It differs from the move constructor in that it:

- Does NOT perform the construction of the object
- Returns a reference to an object
- (notice the destructor is called after the assignment. We captured the data with the “move” operation)

*a constructed object that is temporary. rvalues can not be the recipient of an assignment.
Array2D: move assignment operator

Array2D& Array2D::operator = (Array2D&& arr) {
    name = arr.name;
    rows = arr.rows;
    columns = arr.columns;
    count = arr.count;
    myInts = arr.myInts; // copies pointer
    arr.myInts = nullptr; // severs ties with data
    return *this;
}

Array2D: move assignment operator

Array2D& Array2D::operator = (Array2D&& arr) {
    name = arr.name;
    rows = arr.rows;
    columns = arr.columns;
    count = arr.count;
    myInts = arr.myInts; // copies pointer
    arr.myInts = nullptr; // severs ties with data
    return *this;
}

“&&” is an rvalue reference.
C++ Templates

Templates give us the ability to write code that is independent of types. That is, it allows us to write code that will operate the same way on multiple types including those that we have created. This allows us to do “generic programming”.

If you recall from a previous lecture, templates use the pre-process stage of the compilation process to write code for you.
You may now be familiar with templates such as the vector. The vector provides the same functionality regardless of type.

```cpp
vector<int> int_vect;

vector<float> float_vect;
```
C++ Templates

Templates are declared with the forms:

Function:

    template <class type> return-type function-name(parameter list) {
        // body
    }

Class:

    template <class type> class class-name {
        // body
    }
C++ Templates

Templates allow us to write code in a fill-in-the-blank format. Let’s look at an example using our Array2D class which allows us to interact with a set of integers arranged by rows and columns.

How do we make this work so that we can have rows and columns of any type?

```cpp
template <class T> class Array2DTemplate {}
```

What part(s) of Array2D do we need to redefine?
What part(s) of Array2D do we need to re-write?
What needs to change (minimally):

private:
    int at(int location); // why does this need to be changed?

public:
    ● All constructors
    ● All assignment operator overloads
    ● destructor?

    int at(int row, int column);

    void setValue(int row, int column, int val);
What needs to change (minimally):

Do all (or any) of these “int” types in this constructor need to be changed?

Array2D(int iRows, int iColumns): count(iRows*iColumns), rows(iRows), columns(iColumns)

Why or why not?
What needs to change (minimally):

Array2D::Array2D(int iRows, int iColumns): count(iRows*iColumns), rows(iRows), columns(iColumns) {

    myInts = new <T>[count];

    // this can not stay as it is:
    for (int i=0; i<count; i++)
        myInts[i] = count-i;
}

What needs to change (minimally):

```plaintext
int at(int location);  
Becomes:  <T> at(int location);

int at(int row, int column);  
Becomes:  <T> at(int row, int column);

void setValue(int row, int column, int val);
Becomes:  void setValue(int row, int column, <T> val);
```

From a practical standpoint, the return types in the first two would be problematic.