MTH 4300/4299: Programming and Computer Science II

Lecture 15: Operator Overloading

Jaime Abbariao

▒▒ What is Operator Overloading?

Right now, to compare two `DynamicArray` objects, we'd write something like:

```
bool arrays_equal(const DynamicArray& a, const DynamicArray& b) {
  if (a.get_size() != b.get_size()) return false;
  for (int i = 0; i < a.get_size(); i++) {
    if (a.get(i) != b.get(i)) return false;
  }
  return true;
}
```

Wouldn't it be nicer to just write `a == b`?

Operator overloading lets you define what operators like `==`, `+`, `[]`, and `<<` mean for your own classes.

You've already seen one: `operator=` (the assignment operator) from Lectures 13-14!

▒▒ Operator Overloading Syntax

The general pattern is a function named `operator` followed by the symbol:

```
// Inside the class:
ReturnType operatorSYMBOL(parameters) {
  // implementation
}
```

Some operators you can overload:

Operator	Name	Example
`==`	Equality	`a == b`
`!=`	Inequality	`a != b`
`[]`	Subscript	`a[0]`
`+`	Addition	`a + b`
`<<`	Stream insertion	`std::cout << a`
`+=`	Compound assignment	`a += b`

Some you cannot overload: `::`, `.`, `.*`, `?:`

Let's implement these one at a time, starting with the simplest.

▒▒▒ operator== — Your First Overloaded Operator

Two `DynamicArray`s are equal if they have the same size and the same elements:

```
class DynamicArray {
  // ... (private members: data, size, capacity)
public:
  bool operator==(const DynamicArray& other) const {
    if (size != other.size) {
      return false;
    }
    for (int i = 0; i < size; i++) {
      if (data[i] != other.data[i]) {
        return false;
      }
    }
    return true;
  }
};
```

Now we can write:

```
if (a == b) {
  std::cout << "Arrays are equal!" << std::endl;
}
```

When the compiler sees `a == b`, it calls `a.operator==(b)`.

▒▒▒ operator== — Key Details

Notice the signature: `bool operator==(const DynamicArray& other) const`

- Returns `bool` — comparisons are true/false
- Parameter is `const DynamicArray&` — we only read from `other`
- The function itself is `const` — comparing doesn't modify `*this` either

operator!= can be defined in terms of `==`:

```
bool operator!=(const DynamicArray& other) const {
  return !(*this == other);
}
```

`*this` is the current object. So `!(*this == other)` calls our `operator==` and negates it.

Note: In C++20, the compiler can auto-generate `!=` from `==` using the spaceship operator (`<=>`). We're using C++17, so we write both.

▒▒▒ operator[] — Subscript Operator

Instead of `arr.get(i)`, we want `arr[i]`. And we want assignment to work: `arr[0] = 42`.

```
int& operator[](int index) {
  if (index < 0 || index >= size) {
    std::cout << "Index " << index << " out of bounds!" << std::endl;
    return data[0];  // fallback (not ideal, but simple)
  }
  return data[index];
}
```

Why return `int&` (a reference)?

```
arr[0] = 42;  // This modifies the actual element inside the array
```

If we returned `int` (by value), `arr[0] = 42` would modify a copy — the original array wouldn't change. Returning a reference lets the caller write through it.

▒▒▒ operator[] — The const Version

What happens if someone passes a `DynamicArray` by `const` reference?

```
void print_first(const DynamicArray& arr) {
  std::cout << arr[0] << std::endl;  // ERROR!
}
```

The compiler won't let you call the non-const `operator[]` on a `const` object — it can't guarantee you won't modify it through the returned `int&`.

Fix: provide a `const` overload:

```
const int& operator[](int index) const {
  if (index < 0 || index >= size) {
    std::cout << "Index " << index << " out of bounds!" << std::endl;
    return data[0];
  }
  return data[index];
}
```

- Returns `const int&` — the caller can read but not modify
- The function is `const` — can be called on `const` objects

Rule: the compiler picks the `const` version for `const` objects, the non-const version for non-const objects.

operator[] — Both Versions Together

```cpp
class DynamicArray {
public:
  // Non-const version: called on non-const objects
  // Allows reading AND writing
  int& operator[](int index) {
    return data[index];
  }

  // Const version: called on const objects
  // Allows reading only
  const int& operator[](int index) const {
    return data[index];
  }
};
```

```cpp
DynamicArray arr(5);
arr.push_back(10);
arr[0] = 42;              // non-const version — writes through reference

const DynamicArray& ref = arr;
int val = ref[0];         // const version — read only
// ref[0] = 99;           // ERROR: can't assign to const int&
```

Warning: if the array reallocates (e.g., after `push_back`), any references from `operator[]` become dangling. Be careful not to hold onto them across modifications.

▒▒ operator<< — The Problem

We keep writing `arr.print()`. It would be much nicer to write:

```
std::cout << arr << std::endl;
```

Can we make `<<` a member function of `DynamicArray`? Let's think about what that would mean.

When the compiler sees `a.operator<<(b)`, the left operand is `a` and the right operand is `b`.

For `std::cout << arr`, the left operand is `std::cout` (an `std::ostream`), not our array.

If we made it a member of `DynamicArray`, the call would be:

```
arr.operator<<(std::cout);   // i.e., arr << std::cout
```

That's backwards! We'd have to write `arr << std::cout` instead of `std::cout << arr`.

We can't add a member function to `std::ostream` (it's not our class). So we need a different approach.

▒▒▒ operator<< — Friend Functions

The solution: make it a non-member function that takes both operands as parameters:

```
std::ostream& operator<<(std::ostream& os, const DynamicArray& arr) {
  os << "[";
  for (int i = 0; i < arr.size; i++) {
    os << arr.data[i];
    if (i < arr.size - 1) {
      os << ", ";
    }
  }
  os << "]";
  return os;
}
```

Problem: this function accesses `arr.size` and `arr.data`, which are private.

Solution: declare it as a `friend` inside the class:

```
class DynamicArray {
  friend std::ostream& operator<<(std::ostream& os, const DynamicArray& arr);
  // ...
};
```

`friend` means: "this non-member function is allowed to access my private members."

▒▒ operator<< — Key Details

```
class DynamicArray {
  friend std::ostream& operator<<(std::ostream& os,
                                  const DynamicArray& arr);
private:
  int* data;
  int size;
  int capacity;
public:
  // ... constructors, etc.
};

// Defined OUTSIDE the class (it's not a member!)
std::ostream& operator<<(std::ostream& os, const DynamicArray& arr) {
  os << "[";
  for (int i = 0; i < arr.size; i++) {
    os << arr.data[i];
    if (i < arr.size - 1) os << ", ";
  }
  os << "]";
  return os;
}
```

Why return `std::ostream&`? For chaining:

```
std::cout << arr1 << " and " << arr2 << std::endl;
//         ^^^^^^ returns std::cout
//                        ^^^^^^^^^ returns std::cout again
```

Each `<<` returns the stream so the next `<<` can use it.

▒▒▒ operator+ — Concatenation

We want `a + b` to produce a new array containing all elements of `a` followed by all elements of `b`:

```
DynamicArray operator+(const DynamicArray& other) const {
  DynamicArray result(size + other.size);
  for (int i = 0; i < size; i++) {
    result.push_back(data[i]);
  }
  for (int i = 0; i < other.size; i++) {
    result.push_back(other.data[i]);
  }
  return result;
}
```

Key: this returns a `DynamicArray` by value — a brand new object. It does NOT modify `*this` or `other`.

```
DynamicArray a(3), b(3);
a.push_back(1); a.push_back(2);
b.push_back(3); b.push_back(4);

DynamicArray c = a + b;
std::cout << c;  // [1, 2, 3, 4]
// a and b are unchanged
```

▒▒▒ operator+ — What Happens When It Returns?

Let's trace `DynamicArray c = a + b;` step by step:

```
Step 1: a.operator+(b) is called
        → Inside the function, a local "result" is created: [1, 2, 3, 4]

Step 2: "return result;" — result is about to die (it's local)
        → The compiler has two options:

        Option A (RVO): construct "c" directly where "result" lives
                        No copy, no move — the compiler is smart enough
                        to build result directly in c's memory

        Option B (Move): move constructor transfers result's pointer to c
                         result.data → nullptr (safe to destroy)
                         c.data → [1, 2, 3, 4] (stolen!)

Step 3: The local "result" is destroyed
        → With RVO: nothing to destroy (result IS c)
        → With move: delete[] nullptr (safe, does nothing)
```

In practice: modern compilers almost always use RVO. But your move constructor is the safety net when RVO can't apply.

This is why we spent two lectures on copy/move semantics — they power `operator+`!

▒▒▒ operator+ — Seeing the Move Constructor Fire

To observe whether the move constructor actually runs:

```
// Add a print to the move constructor temporarily:
DynamicArray(DynamicArray&& other)
  : data(other.data), size(other.size), capacity(other.capacity) {
  std::cout << "Move constructor called!" << std::endl;
  other.data = nullptr;
  other.size = 0;
  other.capacity = 0;
}
```

```
DynamicArray c = a + b;  // Does it print "Move constructor called!"?
```

With a modern compiler: probably not (RVO kicks in).

Compile with `-fno-elide-constructors` to disable RVO:

```
g++ -std=c++17 -fno-elide-constructors -o out main.cpp
```

Now you'll see "Move constructor called!" — the compiler is forced to move instead of eliding.

▒▒▒ operator+= — Compound Assignment

`operator+` creates a new array. What if we want to append in-place?

```
DynamicArray& operator+=(const DynamicArray& other) {
  for (int i = 0; i < other.size; i++) {
    push_back(other.data[i]);
  }
  return *this;
}
```

```
DynamicArray a(3), b(3);
a.push_back(1); a.push_back(2);
b.push_back(3); b.push_back(4);

a += b;
std::cout << a;  // [1, 2, 3, 4]  — a was modified
std::cout << b;  // [3, 4]        — b is unchanged
```

Key differences from `operator+`:

	`operator+`	`operator+=`
Modifies `*this`?	No	Yes
Returns	New object (by value)	`*this` (by reference)
Creates a copy?	Yes	No

▒▒▒ operator+= — Implementing operator+ in Terms of +=

A common C++ idiom: write `operator+=` first, then build `operator+` from it.

```
// operator+= does the real work
DynamicArray& operator+=(const DynamicArray& other) {
  for (int i = 0; i < other.size; i++) {
    push_back(other.data[i]);
  }
  return *this;
}

// operator+ creates a copy and uses +=
DynamicArray operator+(const DynamicArray& other) const {
  DynamicArray result(*this);  // copy constructor
  result += other;             // reuse operator+=
  return result;
}
```

Why is this better? The logic for appending is written once in `operator+=`. `operator+` just adds the "make a copy first" step. Less code duplication, fewer bugs.

Why return `DynamicArray&` from `+=`? For chaining:

```
a += b += c;  // c is appended to b, then b (now larger) is appended to a
```

▒▒ Comparison Operators — Lexicographic Order

Just like strings are compared letter-by-letter, we can compare arrays element-by-element.

`{1, 2, 3}` < `{1, 2, 4}` because at the first difference (index 2), `3 < 4`.

`{1, 2}` < `{1, 2, 3}` because the shorter array is a prefix of the longer one.

```
bool operator<(const DynamicArray& other) const {
  int min_size = (size < other.size) ? size : other.size;
  for (int i = 0; i < min_size; i++) {
    if (data[i] < other.data[i]) return true;
    if (data[i] > other.data[i]) return false;
  }
  // All compared elements are equal — shorter array is "less"
  return size < other.size;
}
```

This is the same algorithm that `std::string` uses for comparison!

▒▒ Comparison Operators — Building from operator<

Just like we built `!=` from `==`, we can define all comparison operators from `<`:

```
bool operator>(const DynamicArray& other) const {
  return other < *this;
}

bool operator<=(const DynamicArray& other) const {
  return !(other < *this);
}

bool operator>=(const DynamicArray& other) const {
  return !(*this < other);
}
```

The pattern: pick one "base" operator and define the rest in terms of it.

Expression	Equivalent
`a > b`	`b < a`
`a <= b`	`!(b < a)`
`a >= b`	`!(a < b)`

```
DynamicArray a(2), b(2);
a.push_back(1); a.push_back(2);
b.push_back(1); b.push_back(3);

std::cout << (a < b);   // 1 (true — 2 < 3 at index 1)
std::cout << (a >= b);  // 0 (false)
```

▒▒ operator++ — Prefix and Postfix Increment

For `DynamicArray`, let's define `++` to increment every element by 1.

Problem: `++a` (prefix) and `a++` (postfix) are both spelled `operator++`. How does the compiler tell them apart?

Answer: a dummy `int` parameter that is never used:

```
// Prefix: ++arr  (increment, then return the modified object)
DynamicArray& operator++() {
  for (int i = 0; i < size; i++) {
    data[i]++;
  }
  return *this;
}

// Postfix: arr++  (save old value, increment, return the OLD value)
DynamicArray operator++(int) {  // the 'int' is just a tag
  DynamicArray old(*this);      // copy the current state
  for (int i = 0; i < size; i++) {
    data[i]++;
  }
  return old;                   // return the copy (pre-increment state)
}
```

The `int` parameter is never named and never passed by you — the compiler handles it automatically.

operator++ — Prefix vs Postfix in Action

```
DynamicArray a(3);
a.push_back(1); a.push_back(2); a.push_back(3);

DynamicArray b = ++a;  // a is incremented FIRST, then b gets the result
std::cout << a;  // [2, 3, 4]
std::cout << b;  // [2, 3, 4]  — same as a

DynamicArray c(3);
c.push_back(10); c.push_back(20);

DynamicArray d = c++;  // d gets the OLD value, THEN c is incremented
std::cout << c;  // [11, 21]
std::cout << d;  // [10, 20]  — the pre-increment copy
```

Why is prefix more efficient?

- Prefix (`++a`): modifies in place, returns `*this` by reference. No copy.
- Postfix (`a++`): must create a copy of the old state before modifying. Extra work.

This is why you'll often see `++i` preferred over `i++` in C++ loops — for integers the compiler optimizes it away, but for objects the copy can be real.

Summary

Operator	Signature	Member or Friend?	Returns	Why that return type?
`==`	`bool operator==(const DA&) const`	Member	`bool`	Comparison is true/false
`!=`	`bool operator!=(const DA&) const`	Member	`bool`	Comparison is true/false
`[]`	`int& operator[](int)`	Member	`int&`	So `arr[i] = x` modifies the element
`[]` const	`const int& operator[](int) const`	Member	`const int&`	Read-only access for const objects
`<<`	`ostream& operator<<(ostream&, const DA&)`	Friend	`ostream&`	Chaining: `cout << a << b`
`+`	`DA operator+(const DA&) const`	Member	`DA` (by value)	Creates a new object
`+=`	`DA& operator+=(const DA&)`	Member	`DA&`	Mutates in place, enables chaining
`<`	`bool operator<(const DA&) const`	Member	`bool`	Lexicographic comparison
`++` (pre)	`DA& operator++()`	Member	`DA&`	Modifies and returns `*this`
`++` (post)	`DA operator++(int)`	Member	`DA` (by value)	Returns old copy, then modifies

Patterns to remember:

- Comparisons (`==`, `!=`, `<`, `>`, `<=`, `>=`): return `bool`, take `const&`, function is `const`
- Subscript (`[]`): return reference for read/write access
- Stream (`<<`): must be friend (left operand is `ostream`), return `ostream&`
- Arithmetic (`+`): return by value (new object), triggers move semantics
- Compound assignment (`+=`): mutates `*this`, returns `*this` by reference
- Prefix increment (`++`): mutates and returns `*this` by reference — efficient
- Postfix increment (`++(int)`): copies first, then mutates — the `int` is just a tag

▒▒ Lab: Add Operators to StringHolder

Remember `StringHolder` from the L13 and L14 labs? Let's add operators to it.

```
#include <iostream>
#include <cstring>

class StringHolder {
  friend std::ostream& operator<<(std::ostream& os,
                                  const StringHolder& sh);
private:
  char* data;
  int len;
public:
  StringHolder(const char* str) {
    len = std::strlen(str);
    data = new char[len + 1];
    std::strcpy(data, str);
  }
  ~StringHolder() { delete[] data; }

  // Copy constructor & copy assignment (from L13)
  // Move constructor & move assignment (from L14)
  // ... (same as previous labs — see starter file)
```

```
    int length() const { return len; }

    // TODO: operator==
    // TODO: operator[]  (non-const and const versions)
    // TODO: operator+   (concatenation — returns new StringHolder)
    // TODO: operator+=  (append in place — returns *this)
    // TODO: operator<   (lexicographic comparison)
    // TODO: operator++  (prefix — increment each char by 1)
    // TODO: operator++  (postfix — increment each char by 1, return old)
};

// TODO: operator<< (friend, defined outside class)
```

That's 7 operators to implement! Work through them one at a time.

```cpp
int main() {
  StringHolder s1("Hello");
  StringHolder s2("Hello");
  StringHolder s3("World");

  // Test operator==
  std::cout << "s1 == s2: " << (s1 == s2) << std::endl;  // 1
  std::cout << "s1 == s3: " << (s1 == s3) << std::endl;  // 0

  // Test operator[]
  std::cout << "s1[0]: " << s1[0] << std::endl;           // H
  s1[0] = 'J';
  std::cout << "After s1[0]='J': " << s1 << std::endl;    // Jello

  // Test operator+ and operator<<
  StringHolder s4 = s1 + s3;
  std::cout << "s1 + s3: " << s4 << std::endl;            // JelloWorld
  std::cout << "s2 + s3: " << (s2 + s3) << std::endl;     // HelloWorld

  // Test operator+=
  s1 += s3;
  std::cout << "After s1 += s3: " << s1 << std::endl;     // JelloWorld
```

```
    // Test operator<
    std::cout << "s2 < s3: " << (s2 < s3) << std::endl;     // 1 (H < W)

    // Test operator++ (prefix)
    StringHolder s5("abc");
    ++s5;
    std::cout << "++s5: " << s5 << std::endl;                // bcd

    // Test operator++ (postfix)
    StringHolder s6("xyz");
    StringHolder s7 = s6++;
    std::cout << "s6 after s6++: " << s6 << std::endl;       // yz{
    std::cout << "s7 (old s6):   " << s7 << std::endl;       // xyz

    return 0;
}
```

▒▒ Lab: Expected Output

```
s1 == s2: 1
s1 == s3: 0
s1[0]: H
After s1[0]='J': Jello
s1 + s3: JelloWorld
s2 + s3: HelloWorld
After s1 += s3: JelloWorld
s2 < s3: 1
++s5: bcd
s6 after s6++: yz{
s7 (old s6):   xyz
```

Hints:

- `operator==`: compare `len` first, then use `std::strcmp(data, other.data) == 0`
- `operator[]`: return `char&` (not `int&` — this holds characters). Don't forget the `const` version.
- `operator+`: allocate `len + other.len + 1` characters for the new string. Use `std::strcpy` and `std::strcat`.
- `operator<<`: `os << sh.data;` — that's it! (But it must be a friend to access `data`.)
- `operator+=`: similar to `operator+`, but modify `*this` instead of creating a new object. Delete old `data`, allocate new buffer, copy both strings.
- `operator<`: just use `std::strcmp(data, other.data) < 0`
- `operator++` (prefix): loop through `data` and increment each character. Return `*this`.
- `operator++` (postfix): takes `int` dummy. Copy `*this` first, increment, return the copy.

Compile: `g++ -std=c++17 -o out stringholder.cpp`