“new” Operator Operator Overloading
Keyword: static

Variables

Class members
Keyword: static (variables)

Static variables:

```c
static float my_float;
```

Means that memory is allocated for the lifetime of the application. It has a permanent home.

Non-static variables have a place on the stack and only “exist” within their scope.
Keyword: static (variables)

Static variables:

    static float my_float;

“memory is allocated for the lifetime of the application. It has a permanent home.”

How and why is this useful?
Keyword: static (variables)

Static variables:

    static float my_float;
    static int number_of_uses;

“memory is allocated for the lifetime of the application. It has a permanent home.”

How and why is this useful? You can keep track of a value even after having exited the scope of the function.
Keyword: static (variables)

Static variables:

    static float my_float;
    static int number_of_uses = 0;

There can be only one! These two static variables can only exist once per context. That is, every instance of a class will share the exact same static variables. (The same names can still be declared in another scope by another class.)
Keyword: static (functions)

Static functions:

```cpp
Class MyClass {
    public:
    static void MyFunction();
}
```

Can only access the static variables of the class. Useful for utility-type functions.
Keyword: const (variables)

const int unchangeable_int;

const indicates that a value may not be changed.*

*The compiler will restrict access as we showed on Monday even if we find a way to point to the address of the const. This underscores the value of declaring some variables const.
Keyword: const (functions)

```cpp
const Class_Name Object_name;
```

- const functions can be called on any type of object (const objects or non-const objects)
- Non-const functions can only be called by non-const objects.
- A const OBJECT needs to be initialized at the time of declaration (which can only be done with constructors). (see MyArray constructor)
Topics

- Heap memory vs Stack memory
- “New” operator
- Destructor
- Operator Overloading
- Friend Class
Heap memory vs Stack memory

- When your application begins execution, memory is set aside for variables.
- As variables come into scope, space is allocated for them on the stack.
- Static variables are allocated in their own space.
- “new” causes memory to be allocated on the heap or “free store” (Nowak).
Stack Memory

Stack memory is allocated for:

- Passing parameters
- Local variables

It is ethereal. That is, it disappears when you are finished using it. That is why you can not use something that has gone out of scope. It is not merely something to complicate matters nor to conveniently allow you to re-use variable names.
Heap Memory

Memory allocated on the heap using “new” is available until it is released with the delete operator.

```c++
    int my_int = new int;
    delete my_int;
```

*All memory is released when your application terminates.

**This use of the word “heap” has nothing to do with the data structure.
“new” operator

should only be used if the data object should remain in memory until delete is called.
What’s the Difference?

```c
int stackInts[count];
int* heapInts = new int[count];
```
Dynamic Allocation

```c
int* getMyStackInt(int count) {
    int stackInts[count];
    for (int i=0; i<count; i++)
        stackInts[i] = 100*count-i;
    return(stackInts);
}
```
Dynamic Allocation Using “new”

```c
int* getMyHeapInt(int count) {
    int* heapInts = new int[count];
    for (int i=0; i<count; i++)
        heapInts[i] = count-i;
    return(heapInts);
}
```
Destructor

MyArray::~MyArray(){
    cout << "MyArray destructor: " << count << endl;
    delete[] myInts;
}
Questions

Why can’t we delete

```c
    delete my_2nd_array;
```

Why does this:

```c
    delete[] my_int_arr_ptr;
```

cause an error?
Operator Overloading

C++ allows you to specify more than one definition for a function name or an operator in the same scope.

This is called function overloading and operator overloading respectively.
When Would One Overload an Operator?

What does this mean?

\[ \text{Obj3} = \text{Obj1} + \text{Obj2}; \]
When Would One Overload an Operator?

What does this mean?

\[ \text{Obj3} = \text{Obj1} + \text{Obj2}; \]

It has no meaning until it has been defined.
Function Overloading

MyArray();

MyArray(int count);

Notice that these two function declarations share a name but have different parameter signatures.
Operator Overloading - “<<”

A useful operator that you may want to overload is “<<”.
What is a Friend Class?

- prototypes for friend functions appear in the class definition, friends are not member functions
- defined outside that class' scope
- can access all private and protected members of the class.
appear in the class definition

class MyArray {

public:

    MyArray();

    MyArray(int count);

    ~MyArray();

    int at(int location);

    int size();

    friend std::ostream& operator << (std::ostream& os, MyArray& arr);

};
defined outside that class' scope (MyArray:: not present)

```cpp
std::ostream& operator << (std::ostream& os, MyArray& arr)
{
    for (int j=0; j<arr.count; j++){
        os<<arr.at(j)<<"  \\
    }
    os<<endl;
    os<<endl;
    return os;
}
```
Operator Overloading - "<<" (friend - "count" is private)

```cpp
std::ostream& operator << (std::ostream& os, MyArray& arr)
{
    for (int j=0; j<arr.count; j++){
        os<<arr.at(j)<<(endl;
    }
    os<<endl;
    return os;
}
```
Operator Overloading - “<<” (not friend - size() is public)

```cpp
std::ostream& operator << (std::ostream& os, MyArray& arr)
{
    for (int j=0; j<arr.size(); j++){
        os<<arr.at(j)<<"  "<<endl;
    }
    os<<endl;
    return os;
}
```
Links

Operator Overloading:

https://www.tutorialspoint.com/cplusplus/cpp_overloading.htm

Friend Class:

https://www.tutorialspoint.com/cplusplus/cpp_friend_functions.htm