CS 32 - Discussion 1B

Week 2 - Dynamic Memory Allocation

Outline:

1. Dynamic (vs static) memory allocation
   - The **new** and **delete** keywords

Outline:

1. Dynamic (vs static) memory allocation
   - The **new** and **delete** keywords

2. Classes
   - Copy constructors.
   - Assignment (operator=).
   - Destructors.

Outline:

1. Dynamic (vs static) memory allocation
   - The **new** and **delete** keywords

2. Classes
   - Copy constructors.
   - Assignment (operator=).
   - Destructors.

3. Classes that use dynamic memory**:**

   Why and how we need to implement the above ourselves (as opposed to using built-in) when our class uses dynamic memory.

Outline:

1. Dynamic (vs static) memory allocation
   - The **new** and **delete** keywords

2. Classes
   - **Copy constructors.**
   - **Assignment (operator=).**
   - **Destructors.**

3. Classes that use dynamic memory**:**

   Why and how we need to implement **the above** ourselves (as opposed to using built-in) when our class uses dynamic memory.

Dynamic Memory Allocation

Static allocation:

int arr[100];

Dynamic allocation:

int* arr = new int[100];

- **new keyword:**

  allocates memory
  calls constructor
  returns pointer to constructed object

Dynamic Memory Allocation

Static allocation:

int arr[100];

Dynamic allocation:

int* arr = new int[100];
delete [ ] arr;

- **new keyword:**

  allocates memory
  calls constructor
  returns pointer to constructed object

- **delete keyword:**

  calls destructor of pointed to object
  frees memory that was allocated by new

Dynamic Memory Allocation

Static allocation:

int arr[100];

Dynamic allocation:

int* arr = new int[100];
delete [ ] arr;

- **new keyword:**

  allocates memory
  calls constructor
  returns pointer to constructed object

- **delete keyword:**

  calls destructor of pointed to object
  frees memory that was allocated by new

- **rule:**

  if new was called, then delete must be called
  (otherwise we get a memory leak)

Classes: Construction and Destruction

```
class Ec
{
    public:
        Ec(); // default constructor 1
        Ec(int data); // default constructor 2...
        Ec(const Ec& other); // copy constructor
        Ec& operator=(const Ec& other); // assignment
        ~Ec(); // destructor

    private:
        string m_name;
        int m_data;
};
```

Classes: Construction and Destruction

```
class Ec
{
    public:
        Ec(); // default constructor 1
        Ec(int data); // default constructor 2…
        Ec(const Ec& other); // copy constructor
        Ec& operator=(const Ec& other); // assignment
        ~Ec(); // destructor

    private:
        string m_name;
        int m_data;
};
```

- **Note**: can define as many default constructors as you want, as long as each takes different arguments

Classes: Construction and Destruction

```
class Ec
{
    public:
        Ec(); // default constructor 1
        Ec(int data); // default constructor 2...
        Ec(const Ec& other); // copy constructor
        Ec& operator=(const Ec& other); // assignment
        ~Ec(); // destructor

    private:
        string m_name;
        int m_data;
};
```

```
Ec e1;
Ec e2(10);
Ec e3(e1);
e1 = e3;
```

- **Note**: can define as many default constructors as you want, as long as each takes different arguments

Classes: Construction and Destruction

```
class Ec
{
    public:
        Ec(); // default constructor 1
        Ec(int data); // default constructor 2…
        Ec(const Ec& other); // copy constructor
        Ec& operator=(const Ec& other); // assignment
        ~Ec(); // destructor

    private:
        string m_name;
        int m_data;
};
```

```
Ec e1; // calls default constructor 1
Ec e2(10);
Ec e3(e1);
e1 = e3;
```

- **Note**: can define as many default constructors as you want, as long as each takes different arguments

Classes: Construction and Destruction

```
class Ec
{
    public:
        Ec(); // default constructor 1
        Ec(int data); // default constructor 2…
        Ec(const Ec& other); // copy constructor
        Ec& operator=(const Ec& other); // assignment
        ~Ec(); // destructor

    private:
        string m_name;
        int m_data;
};
```

```
Ec e1; // calls default constructor 1
Ec e2(10); // calls default constructor 2
Ec e3(e1);
e1 = e3;
```

- **Note**: can define as many default constructors as you want, as long as each takes different arguments

Classes: Construction and Destruction

```
class Ec
{
    public:
        Ec(); // default constructor 1
        Ec(int data); // default constructor 2...
        Ec(const Ec& other); // copy constructor
        Ec& operator=(const Ec& other); // assignment
        ~Ec(); // destructor

    private:
        string m_name;
        int m_data;
};
```

```
Ec e1; // calls default constructor 1
Ec e2(10); // calls default constructor 2
Ec e3(e1); // calls copy constructor
e1 = e3;
```

- **Note**: can define as many default constructors as you want, as long as each takes different arguments

Classes: Construction and Destruction

```
class Ec
{
    public:
        Ec(); // default constructor 1
        Ec(int data); // default constructor 2...
        Ec(const Ec& other); // copy constructor
        Ec& operator=(const Ec& other); // assignment
        ~Ec(); // destructor

    private:
        string m_name;
        int m_data;
};
```

```
Ec e1; // calls default constructor 1
Ec e2(10); // calls default constructor 2
Ec e3(e1); // calls copy constructor
e1 = e3; // calls operator=
```

- **Note**: can define as many default constructors as you want, as long as each takes different arguments

Classes: Construction and Destruction

```
class Ec
{
    public:
        Ec(); // default constructor 1
        Ec(int data); // default constructor 2…
        Ec(const Ec& other); // copy constructor
        Ec& operator=(const Ec& other); // assignment
        ~Ec(); // destructor

    private:
        string m_name;
        int m_data;
};
```

```
Ec e1; // calls default constructor 1
Ec e2(10); // calls default constructor 2
Ec e3(e1); // calls copy constructor
e1 = e3; // calls operator=
equivalent to…
e1.operator=(e3);
```

- **Note**: can define as many default constructors as you want, as long as each takes different arguments

Classes: Construction and Destruction

```
class Ec
{
    public:
        Ec(); // default constructor 1
        Ec(int data); // default constructor 2…
        Ec(const Ec& other); // copy constructor
        Ec& operator=(const Ec& other); // assignment
        ~Ec(); // destructor

    private:
        string m_name;
        int m_data;
};
```

- **Note**: can define as many default constructors as you want, as long as each takes different arguments

```
Ec e1; // calls default constructor 1
Ec e2(10); // calls default constructor 2
Ec e3(e1); // calls copy constructor
e1 = e3; // calls operator=
equivalent to…
e1.operator=(e3);
```

Pass-by-value function calls

```
void h(Ec e)
{

    …
    return;
}
```

Classes: Construction and Destruction

```
class Ec
{
    public:
        Ec(); // default constructor 1
        Ec(int data); // default constructor 2…
        Ec(const Ec& other); // copy constructor
        Ec& operator=(const Ec& other); // assignment
        ~Ec(); // destructor

    private:
        string m_name;
        int m_data;
};
```

- **Note**: can define as many default constructors as you want, as long as each takes different arguments

```
Ec e1; // calls default constructor 1
Ec e2(10); // calls default constructor 2
Ec e3(e1); // calls copy constructor
e1 = e3; // calls operator=
equivalent to…
e1.operator=(e3);
```

Pass-by-value function calls

```
void h(Ec e)
{

    …
    return;
}

h(e1);
```

Classes: Construction and Destruction

```
class Ec
{
    public:
        Ec(); // default constructor 1
        Ec(int data); // default constructor 2…
        Ec(const Ec& other); // copy constructor
        Ec& operator=(const Ec& other); // assignment
        ~Ec(); // destructor

    private:
        string m_name;
        int m_data;
};
```

- **Note**: can define as many default constructors as you want, as long as each takes different arguments

```
Ec e1; // calls default constructor 1
Ec e2(10); // calls default constructor 2
Ec e3(e1); // calls copy constructor
e1 = e3; // calls operator=
equivalent to…
e1.operator=(e3);
```

Pass-by-value function calls

```
void h(Ec e)
{
     // calls copy constructor to create local variable e: Example e(e1);
    …
    return;
}

h(e1);
```

Classes: Construction and Destruction

```
class Ec
{
    public:
        Ec(); // default constructor 1
        Ec(int data); // default constructor 2…
        Ec(const Ec& other); // copy constructor
        Ec& operator=(const Ec& other); // assignment
        ~Ec(); // destructor

    private:
        string m_name;
        int m_data;
};
```

- **Note**: can define as many default constructors as you want, as long as each takes different arguments

```
Ec e1; // calls default constructor 1
Ec e2(10); // calls default constructor 2
Ec e3(e1); // calls copy constructor
e1 = e3; // calls operator=
equivalent to…
e1.operator=(e3);
```

Pass-by-value function calls

```
void h(Ec e)
{
     // calls copy constructor to create local variable e: Example e(e1);
    …
    return; // calls destructor to destroy local variable e
}

h(e1);
```

Copy Constructors:
Built-in (compiler-generated)

```
class Ec
{
    public:
        Ec(); // default constructor 1
        Ec(int data); // default constructor 2…
        Ec(const Ec& other); // copy constructor
        Ec& operator=(const Ec& other); // assignment
        ~Ec(); // destructor

    private:
        string m_name;
        int m_data;
};
```

- If you do not implement a copy constructor (or operator=, or destructor), then the compiler generates one for you.
- Built-in CC: simply calls CC of each data member to make a copy

Copy Constructors:
Built-in (compiler-generated)

```
class Ec
{
    public:
        Ec(); // default constructor 1
        Ec(int data); // default constructor 2…
        Ec(const Ec& other); // copy constructor
        Ec& operator=(const Ec& other); // assignment
        ~Ec(); // destructor

    private:
        string m_name;
        int m_data;
};
```

- If you do not implement a copy constructor (or operator=, or destructor), then the compiler generates one for you.
- Built-in CC: simply calls CC of each data member to make a copy

Ec's default constructor

```
Ec::Ec(const Ec& other)
{
   m_name(other.m_name);
   m_data(other.m_data);
}
```

Copy Constructors:
Built-in (compiler-generated)

```
class Ec
{
    public:
        Ec(); // default constructor 1
        Ec(int data); // default constructor 2…
        Ec(const Ec& other); // copy constructor
        Ec& operator=(const Ec& other); // assignment
        ~Ec(); // destructor

    private:
        string m_name;
        int m_data;
};
```

- If you do not implement a copy constructor (or operator=, or destructor), then the compiler generates one for you.
- Built-in CC: simply calls CC of each data member to make a copy

Ec's default constructor

```
Ec::Ec(const Ec& other)
{
   m_name(other.m_name);
   m_data(other.m_data);
}
```

Calls string's CC

Copy Constructors:
Built-in (compiler-generated)

```
class Ec
{
    public:
        Ec(); // default constructor 1
        Ec(int data); // default constructor 2…
        Ec(const Ec& other); // copy constructor
        Ec& operator=(const Ec& other); // assignment
        ~Ec(); // destructor

    private:
        string m_name;
        int m_data;
};
```

- If you do not implement a copy constructor (or operator=, or destructor), then the compiler generates one for you.
- Built-in CC: simply calls CC of each data member to make a copy

Ec's default constructor

```
Ec::Ec(const Ec& other)
{
   m_name(other.m_name);
   m_data(other.m_data);
}
```

Calls string's CC

- **Q**: Would we need to implement these things for the Ec class as is?

Copy Constructors:
Built-in (compiler-generated)

```
class Ec
{
    public:
        Ec(); // default constructor 1
        Ec(int data); // default constructor 2…
        Ec(const Ec& other); // copy constructor
        Ec& operator=(const Ec& other); // assignment
        ~Ec(); // destructor

    private:
        string m_name;
        int m_data;
};
```

- If you do not implement a copy constructor (or operator=, or destructor), then the compiler generates one for you.
- Built-in CC: simply calls CC of each data member to make a copy

Ec's default constructor

```
Ec::Ec(const Ec& other)
{
   m_name(other.m_name);
   m_data(other.m_data);
}
```

Calls string's CC

- **Q**: Would we need to implement these things for the Ec class as is?

A: No, b/c the built-in versions already do the right thing.

Copy Constructors:
Built-in (compiler-generated)

```
class Ec
{
    public:
        Ec(); // default constructor 1
        Ec(int data); // default constructor 2…
        Ec(const Ec& other); // copy constructor
        Ec& operator=(const Ec& other); // assignment
        ~Ec(); // destructor

    private:
        string m_name;
        int m_data;
};
```

- If you do not implement a copy constructor (or operator=, or destructor), then the compiler generates one for you.
- Built-in CC: simply calls CC of each data member to make a copy

Ec's default constructor

```
Ec::Ec(const Ec& other)
{
   m_name(other.m_name);
   m_data(other.m_data);
}
```

Calls string's CC

- **Q**: Would we need to implement these things for the Ec class as is?

A: No, b/c the built-in versions already do the right thing.

- **Q**: When would the built-in versions not do the right thing?

Copy Constructors:
Built-in (compiler-generated)

```
class Ec
{
    public:
        Ec(); // default constructor 1
        Ec(int data); // default constructor 2…
        Ec(const Ec& other); // copy constructor
        Ec& operator=(const Ec& other); // assignment
        ~Ec(); // destructor

    private:
        string m_name;
        int m_data;
};
```

- If you do not implement a copy constructor (or operator=, or destructor), then the compiler generates one for you.
- Built-in CC: simply calls CC of each data member to make a copy

Ec's default constructor

```
Ec::Ec(const Ec& other)
{
   m_name(other.m_name);
   m_data(other.m_data);
}
```

Calls string's CC

- **Q**: Would we need to implement these things for the Ec class as is?

A: No, b/c the built-in versions already do the right thing.

- **Q**: When would the built-in versions not do the right thing?

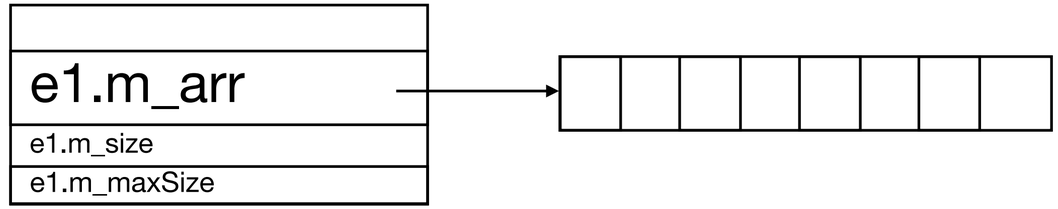
A: When the class uses dynamically allocated memory. (Possibly in other cases when the class has a pointer as a data member).

Copy Constructors:

Classes with Dynamically Allocated Data

The built-in copy constructor simply copies the values of e1's data members

Ec e1;




```
class Ec
{
    public:
        Ec(int maxSize); // default constructor
        Ec(const Ec& other); // copy constructor
        Ec& operator=(const Ec& other); // assignment
        ~Ec(); // destructor

    private:
        int* m_arr;
        int m_size;
        int m_maxSize;
};
```

Copy Constructors:

Classes with Dynamically Allocated Data

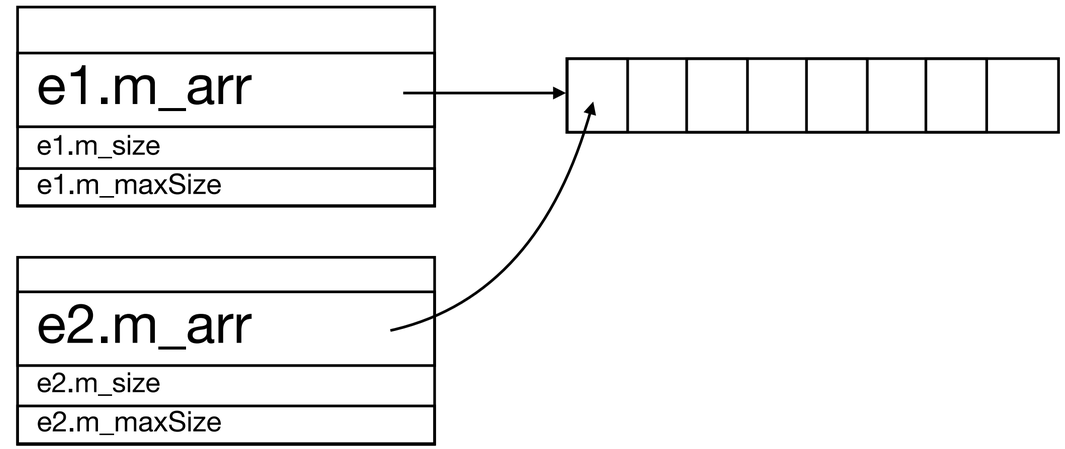
The built-in copy constructor simply copies the values of e1's data members

```
class Ec
{
    public:
        Ec(int maxSize); // default constructor
        Ec(const Ec& other); // copy constructor
        Ec& operator=(const Ec& other); // assignment
        ~Ec(); // destructor

    private:
        int* m_arr;
        int m_size;
        int m_maxSize;
};
```

Ec e1;

Ec e2(e1);



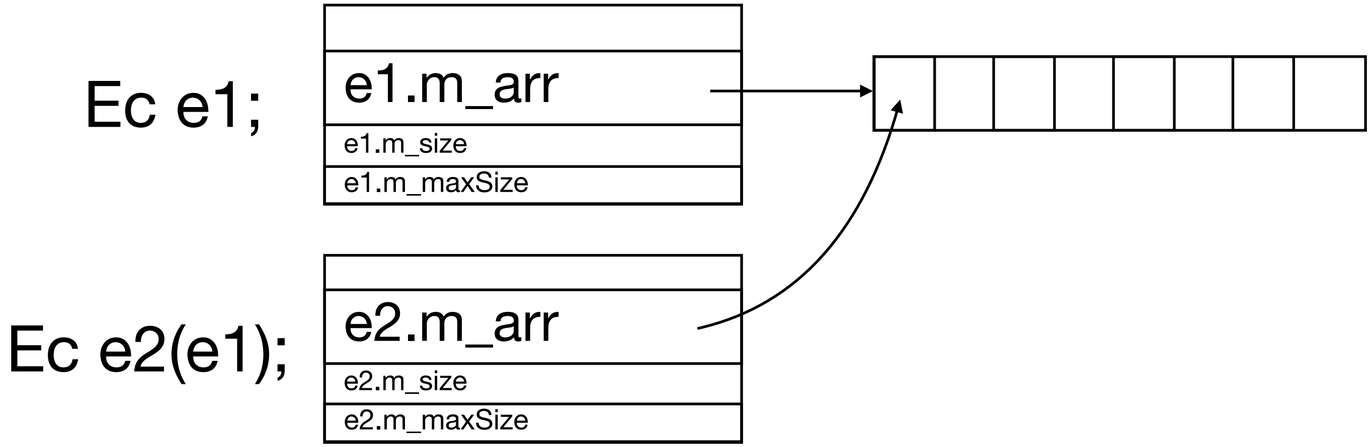
Copy Constructors:

Classes with Dynamically Allocated Data

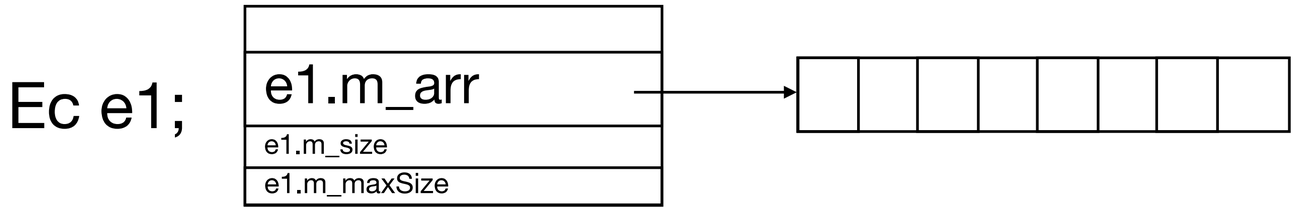
```
class Ec
{
    public:
        Ec(int maxSize); // default constructor
        Ec(const Ec& other); // copy constructor
        Ec& operator=(const Ec& other); // assignment
        ~Ec(); // destructor

    private:
        int* m_arr;
        int m_size;
        int m_maxSize;
};
```

The built-in copy constructor simply copies the values of e1's data members



What we want the copy constructor to do:



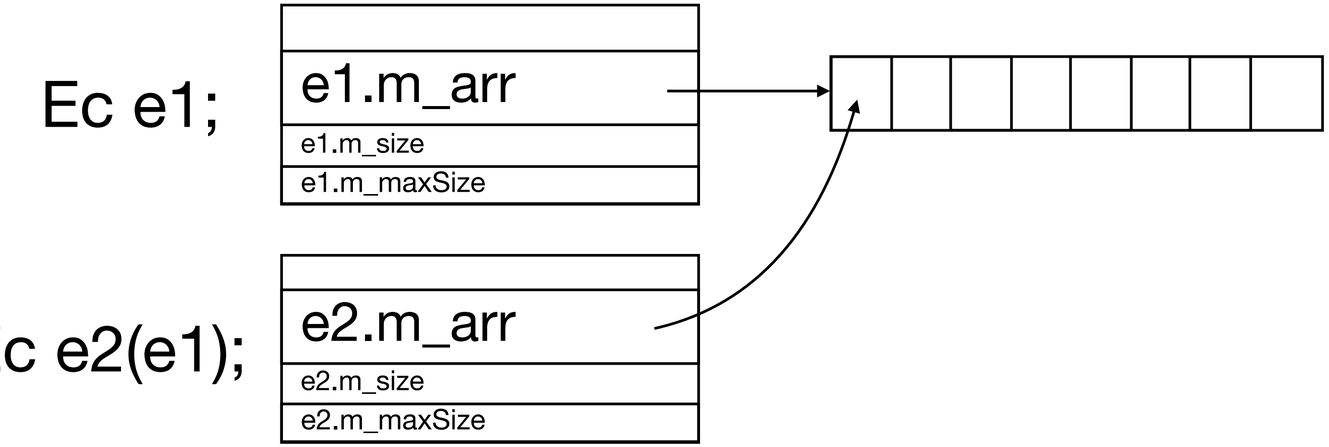
Copy Constructors:

Classes with Dynamically Allocated Data

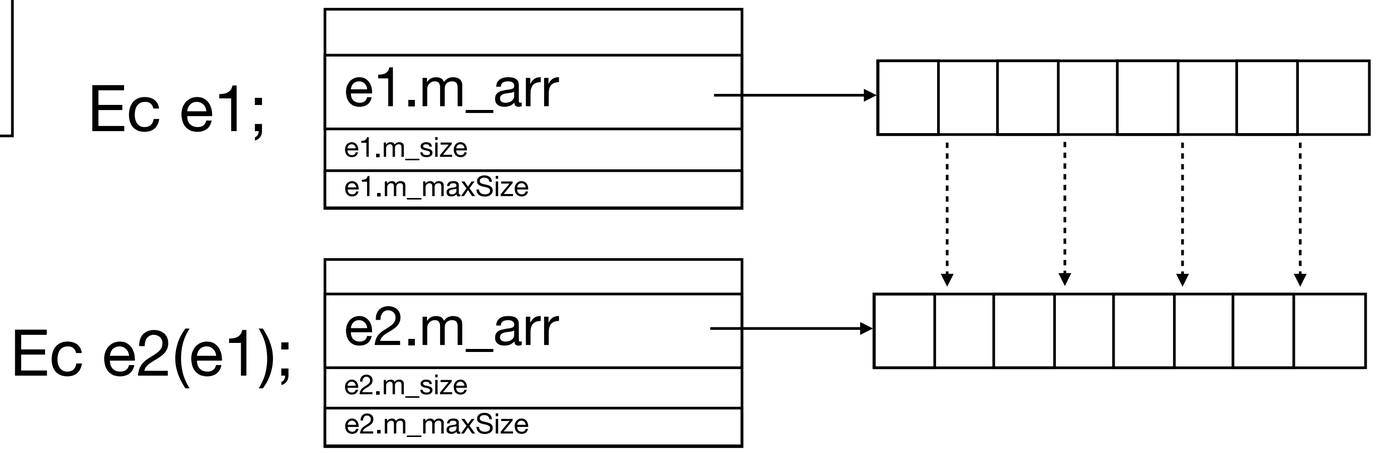
```
class Ec
{
    public:
        Ec(int maxSize); // default constructor
        Ec(const Ec& other); // copy constructor
        Ec& operator=(const Ec& other); // assignment
        ~Ec(); // destructor

    private:
        int* m_arr;
        int m_size;
        int m_maxSize;
};
```

The built-in copy constructor simply copies the values of e1's data members



What we want the copy constructor to do:



Copy Constructors:

Classes with Dynamically Allocated Data

```
class Ec
{
    public:
        Ec(int maxSize); // default constructor
        Ec(const Ec& other); // copy constructor
        Ec& operator=(const Ec& other); // assignment
        ~Ec(); // destructor

    private:
        int* m_arr;
        int m_size;
        int m_maxSize;
};
```

Default constructor:

Copy Constructors:

Classes with Dynamically Allocated Data

```
class Ec
{
    public:
        Ec(int maxSize); // default constructor
        Ec(const Ec& other); // copy constructor
        Ec& operator=(const Ec& other); // assignment
        ~Ec(); // destructor

    private:
        int* m_arr;
        int m_size;
        int m_maxSize;
};
```

Default constructor:

```
Ec::Ec(int maxSize) : m_size(0), m_maxSize(maxSize)
{
        m_arr = new int[m_maxSize];
}
```

Copy Constructors:

Classes with Dynamically Allocated Data

```
class Ec
{
    public:
        Ec(int maxSize); // default constructor
        Ec(const Ec& other); // copy constructor
        Ec& operator=(const Ec& other); // assignment
        ~Ec(); // destructor

    private:
        int* m_arr;
        int m_size;
        int m_maxSize;
};
```

Default constructor:

```
Ec::Ec(int maxSize) : m_size(0), m_maxSize(maxSize)
{
        m_arr = new int[m_maxSize];
}
```

Compiler-generated CC:

```
Ec::Ec(const Ec& other)
{
    m_size(other.m_size);
    m_maxSize(other.m_maxSize);
    m_arr(other.m_arr);
}
```

Copy Constructors:

Classes with Dynamically Allocated Data

```
class Ec
{
    public:
        Ec(int maxSize); // default constructor
        Ec(const Ec& other); // copy constructor
        Ec& operator=(const Ec& other); // assignment
        ~Ec(); // destructor

    private:
        int* m_arr;
        int m_size;
        int m_maxSize;
};
```

Default constructor:

```
Ec::Ec(int maxSize) : m_size(0), m_maxSize(maxSize)
{
        m_arr = new int[m_maxSize];
}
```

Compiler-generated CC:

```
Ec::Ec(const Ec& other)
{
    m_size(other.m_size);
    m_maxSize(other.m_maxSize);
    m_arr(other.m_arr);
}
```

Copy Constructors:

Classes with Dynamically Allocated Data

```
class Ec
{
    public:
        Ec(int maxSize); // default constructor
        Ec(const Ec& other); // copy constructor
        Ec& operator=(const Ec& other); // assignment
        ~Ec(); // destructor

    private:
        int* m_arr;
        int m_size;
        int m_maxSize;
};
```

Default constructor:

```
Ec::Ec(int maxSize) : m_size(0), m_maxSize(maxSize)
{
      m_arr = new int[m_maxSize];
}
```

Compiler-generated CC:

```
Ec::Ec(const Ec& other)
{
    m_size(other.m_size);
    m_maxSize(other.m_maxSize);
    m_arr(other.m_arr);
}
```

Correct CC:

```
Ec::Ec(const Ec& other)
{
    m_size(other.m_size);
    m_maxSize(other.m_maxSize);
    m_arr = new int[m_maxSize];
    for (int i = 0; i < m_size; i++)
        m_arr[i] = other.m_arr[i];
}
```

Destructors:

Classes with Dynamically Allocated Data

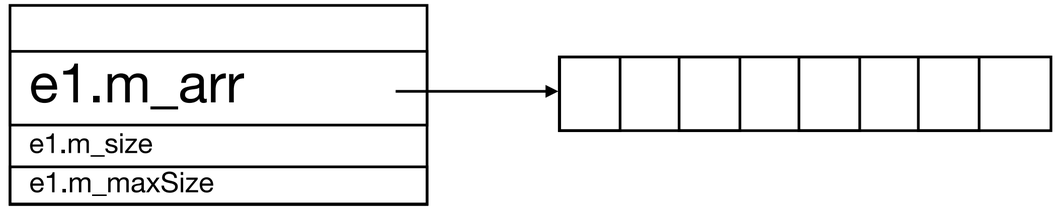
```
class Ec
{
    public:
        Ec(int maxSize); // default constructor
        Ec(const Ec& other); // copy constructor
        Ec& operator=(const Ec& other); // assignment
        ~Ec(); // destructor

    private:
        int* m_arr;
        int m_size;
        int m_maxSize;
};
```

Built-in Destructor

Simply calls destructor for each data member

Ec e1(10);



Default constructor:

```
Ec::Ec(int maxSize) : m_size(0), m_maxSize(maxSize)
{
      m_arr = new int[m_maxSize];
}
```

Destructors:

Classes with Dynamically Allocated Data

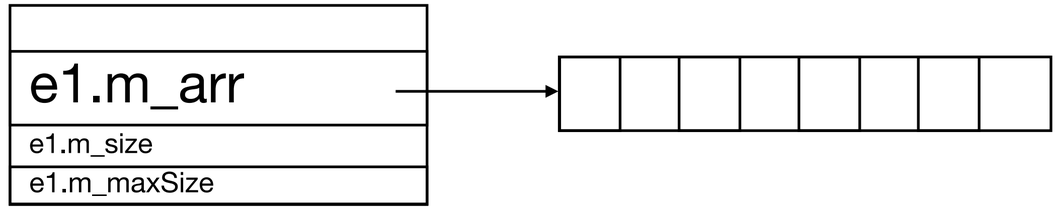
```
class Ec
{
    public:
        Ec(int maxSize); // default constructor
        Ec(const Ec& other); // copy constructor
        Ec& operator=(const Ec& other); // assignment
        ~Ec(); // destructor

    private:
        int* m_arr;
        int m_size;
        int m_maxSize;
};
```

Built-in Destructor

Simply calls destructor for each data member

Ec e1(10);



e1.~Ec();

Default constructor:

```
Ec::Ec(int maxSize) : m_size(0), m_maxSize(maxSize)
{
      m_arr = new int[m_maxSize];
}
```

Destructors:

Classes with Dynamically Allocated Data

```
class Ec
{
    public:
        Ec(int maxSize); // default constructor
        Ec(const Ec& other); // copy constructor
        Ec& operator=(const Ec& other); // assignment
        ~Ec(); // destructor

    private:
        int* m_arr;
        int m_size;
        int m_maxSize;
};
```

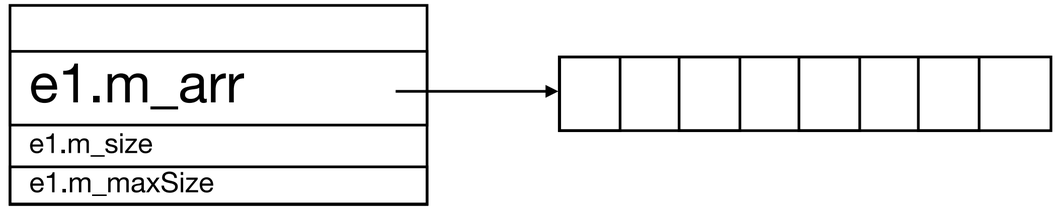
Default constructor:

```
Ec::Ec(int maxSize) : m_size(0), m_maxSize(maxSize)
{
      m_arr = new int[m_maxSize];
}
```

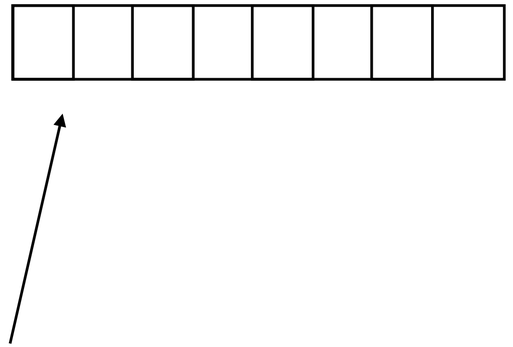
Built-in Destructor

Simply calls destructor for each data member

Ec e1(10);



e1.~Ec();



Need destructor to explicitly deallocate this memory

(Otherwise we have a memory leak)

Destructors:

Classes with Dynamically Allocated Data

```
class Ec
{
    public:
        Ec(int maxSize); // default constructor
        Ec(const Ec& other); // copy constructor
        Ec& operator=(const Ec& other); // assignment
        ~Ec(); // destructor

    private:
        int* m_arr;
        int m_size;
        int m_maxSize;
};
```

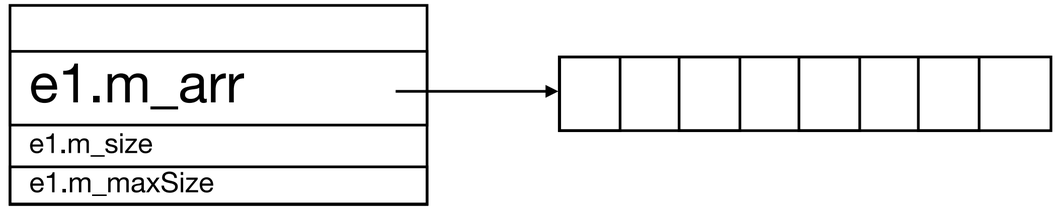
Default constructor:

```
Ec::Ec(int maxSize) : m_size(0), m_maxSize(maxSize)
{
      m_arr = new int[m_maxSize];
}
```

Built-in Destructor

Simply calls destructor for each data member

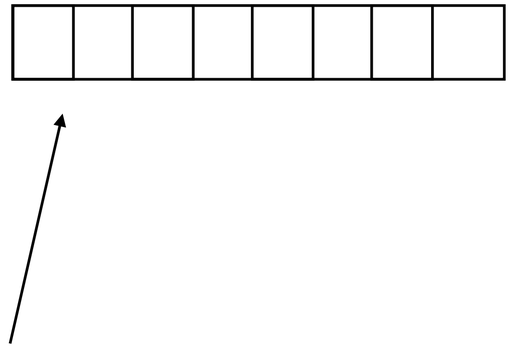
Ec e1(10);



Ec* ep = new Ec(10);

e1.~Ec();

delete ep;



Need destructor to explicitly deallocate this memory

(Otherwise we have a memory leak)

Destructors:

Classes with Dynamically Allocated Data

```
class Ec
{
    public:
        Ec(int maxSize); // default constructor
        Ec(const Ec& other); // copy constructor
        Ec& operator=(const Ec& other); // assignment
        ~Ec(); // destructor

    private:
        int* m_arr;
        int m_size;
        int m_maxSize;
};
```

Default constructor:

```
Ec::Ec(int maxSize) : m_size(0), m_maxSize(maxSize)
{
      m_arr = new int[m_maxSize];
}
```

Built-in Destructor

Simply calls destructor for each data member

Correct Destructor

```
Ec::~Ec()
{
      delete [ ] m_arr;
}
```

Done.

- Resource: Smallberg's example string class.
  (code on course webpage)