CMSC202
Computer Science II for Majors

Lecture 13 –
Friends and More

Dr. Katherine Gibson

Last Class We Covered

- Linked Lists
  - Traversal
  - Creation
  - Insertion
  - Deletion

Any Questions from Last Time?

Today's Objectives

- To cover some miscellaneous topics:
  - Friends
  - Destructors
    - Freeing memory in a structure
  - Copy Constructors
  - Assignment Operators

Friend Functions and Classes

Why Have Friends?

- Giving direct access to private variables is not possible if the function is not a class method
- But using accessors can be cumbersome, especially for something like an overloaded insertion operator (<<)
- Use a “friend” function to give direct access, even though the function is not called on an object of that class

Friend Functions

- Non-member functions that have member-style access
- Function is declared inside the class
  - Will be public regardless of specifier
- Designate using the ***friend*** keyword

```
friend void aFriendFunction();
```

Friend Classes

- Classes can also be declared to be friends of another class

```
class Milo {
public:
   friend class Otis;
};

class Otis { ... };
```

the Otis class now has access to all of the private members of the Milo class

Forward Declarations

- When one class references another in its definition, we need a ***forward declaration***
  - Tell the compiler it exists, without defining it

- In order to reference the **`Otis`** class before it's defined, we need something similar:

```
class Otis;
```

  - before the **`Milo`** class declaration

Using Friends

- Why give access to private member variables?

- Useful for testing functionality
- Increased speed
- Operator overloading
- Enhances encapsulation
  - A function being a friend is specified **in** the class

Destructors

Destructors

- ***Destructors*** are the opposite of constructors
- Used when **`delete()`** is called on an instance of a user-created class
- Compiler automatically provides one for you
  - Does not take into account dynamic memory
  - If your class uses dynamic memory, you must write a better destructor to prevent memory leaks!

Destructor Example: Date

- Let's say we have a new member variable of our **`Date`** class called '**`m_next_holiday`**'
  - Pointer to a string with name of the next holiday

```
class Date {
private:
   int      m_month;
   int      m_day;
   int      m_year;
   string *m_next_holiday ;
};
```

Destructor Example: Date

- We will need to update the constructor

```
Date::Date (int m, int d, int y,
            string next_holiday) {
   SetMonth(m);
   SetDay(d);
   SetYear(y);
   m_next_holiday  = new string;
   *m_next_holiday = next_holiday;
}
```

What other changes do we need to make to a class when adding a new member variable?

Creating a Destructor

- We also now need to create a destructor of our own:

```
~Date();      // our destructor
```

- Destructors must have a tilde at the front
- Similar to a constructor:
  - Destructor has no return type
  - Same name as the class

Basic Destructor Definition

- The destructor needs to free ***all*** of the dynamically allocated memory
  - Otherwise we will have ***memory leaks***

- Most basic version of a destructor

```
Date::~Date() {
   delete m_next_holiday;
}
```

Security and Carefulness

```
Date::~Date() {
  delete m_next_holiday;
}
```

- This works, but it isn't very secure for the data, and it isn't very careful with our pointers
  - What if someone gets access to this memory later?
  - What if my code tries to access m_next_holiday after it's been deleted?

Ideal Destructor Definition

- Clears ***all*** information and sets pointers to **NULL**

```
Date::~Date() {
    // clear member variable info
    m_day = m_month = m_year = 0;
    *m_next_holiday = "";
    // free and set pointers to NULL
    delete m_next_holiday;
    m_next_holiday = NULL;
}
```

Why aren't we using the mutator functions here?

Freeing Memory

- Done using the **`delete()`** function
  - Takes a pointer as an argument:

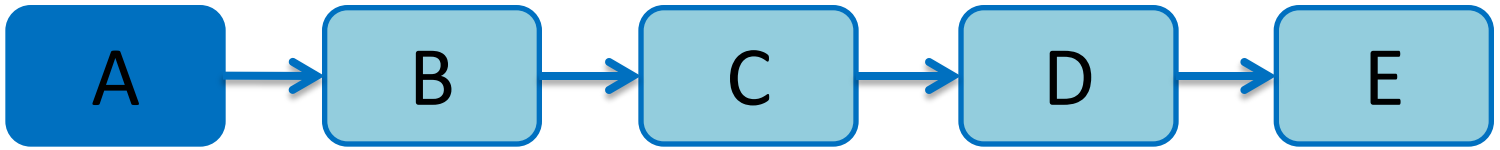
```
delete(grades);

delete(letters);
```

- **`delete()`** does not work recursively
  - For each individual allocation, there **must** be an individual call to free that allocated memory
  - Called in a **sensible** order

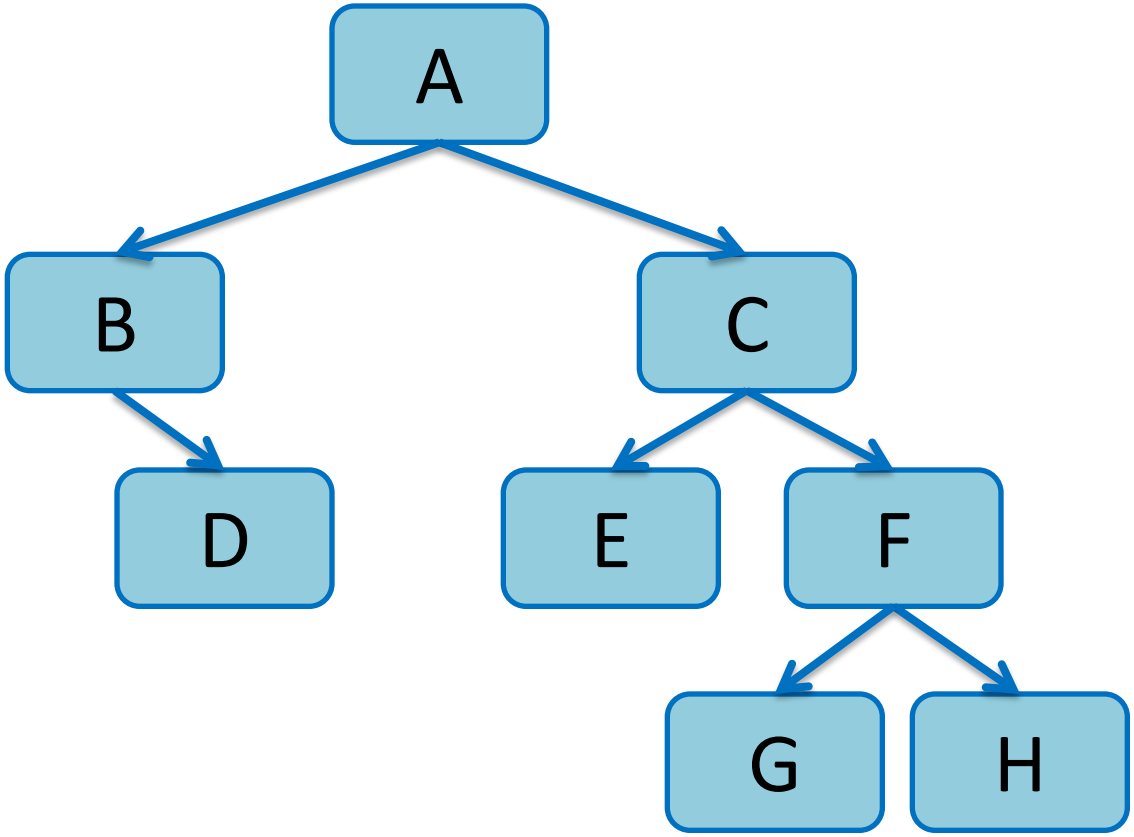
Freeing in Order

In what order would you free the nodes of this linked list?



Freeing in Order

In what order would you free the nodes of this binary tree?



Copy Constructors and Assignment Operators

Copying Objects…

- When does C++ make copies of objects?
  - Pass by value
  - Return by value
  - Assignment
  - and…
  - New object initialized from existing object

Copy Constructor

- Initialize an object based on an existing object
- Examples:

```
int a = 7;
int b(a);  // Copy constructor

Shoe shoeOfMJ( "Nike", 16 );
Shoe myShoe( shoeOfMJ );  // Copy
```

Copy Constructor

- Use when dynamic memory is allocated
- Syntax:
  - Prototype:

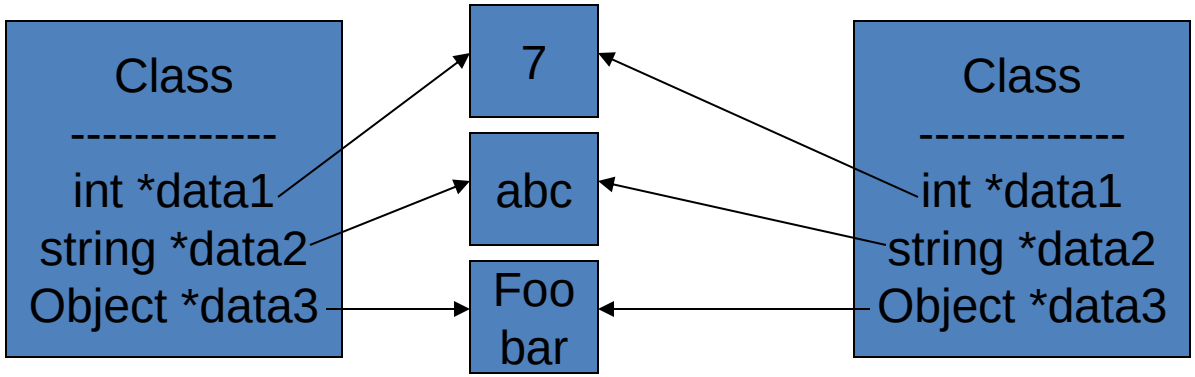
```
ClassName( const ClassName& obj );
```

  - Implementation:

```
ClassName::ClassName( const ClassName& obj )
{
    // code to dynamically allocate data
}
```

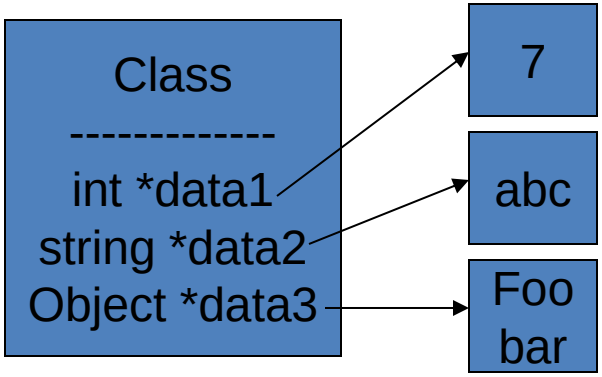
Why do we care?

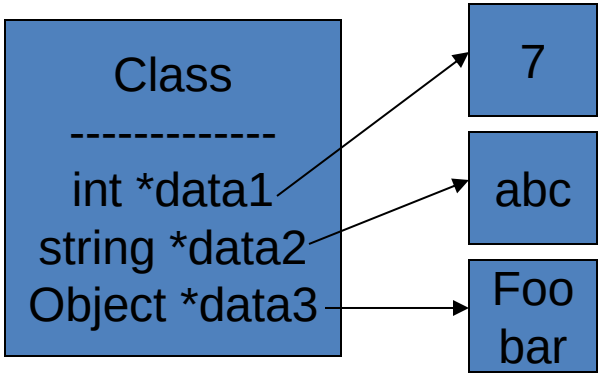
- Remember
  - Assignment (by default) makes a direct copy of data members…
  - With dynamic memory – this would be copying pointers



What do we want?

- Each object should have own memory allocated to members…





Example

```
class Shoe
{
    public:
        Shoe( const Shoe& shoe );
    private:
        int *m_size;
        string *m_brand;
};

Shoe::Shoe( const Shoe& shoe )
{
    m_size = new int( *shoe.m_size );
    m_brand = new string( *shoe.m_brand );
}
```

What's going on here?

What else?

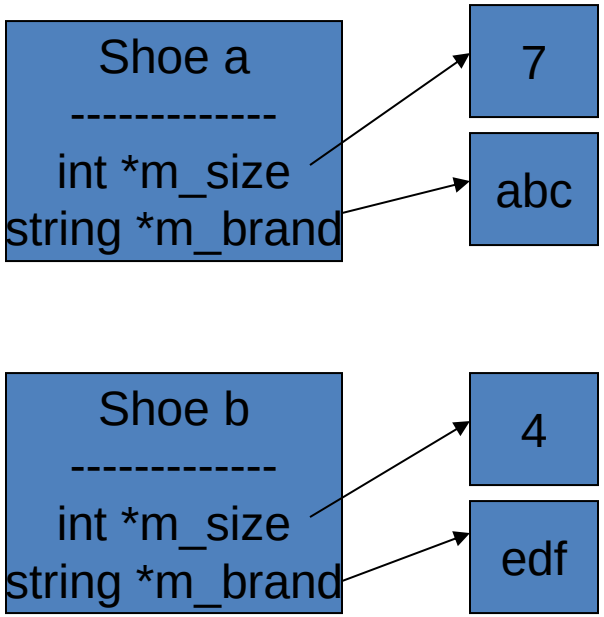
- Assignment Operator
  - Define if using dynamic memory
- Syntax:
  - Prototype:

```
ClassName& operator=( const ClassName& obj );
```

  - Definition:

```
ClassName& ClassName::operator=( const ClassName& obj )
{
    // Deallocate existing memory, if necessary
    // Allocate new memory
}
```

What's Wrong With This?



What happened to the memory b was pointing to first???

```
Shoe& Shoe::operator=(
        const Shoe& shoe )
{
    m_size =
      new int(*shoe.m_size);
    m_brand =
      new string(*shoe.m_brand);
}

// In main()
Shoe a(7, "abc");
Shoe b(4, "edf");

b = a;
```

What's wrong with this?

```
void  Shoe::operator=( const Shoe& shoe )
{
  *m_size = *shoe.m_size;
  *m_brand = *shoe.m_brand;



}

Shoe a(7, "abc");
Shoe b(4, "edf");
Shoe c(9, "ghi");

c = b = a;
```

How does the c = b work, when b = a returns nothing??

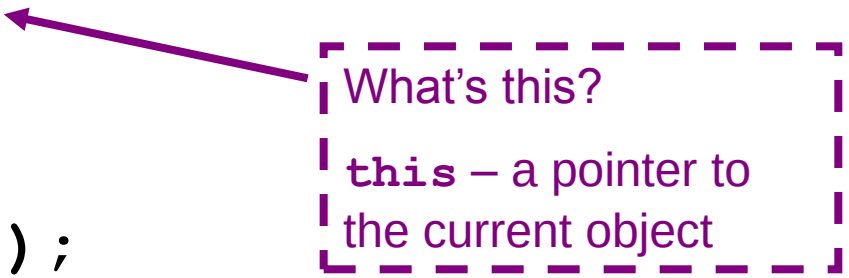
Fixed

```
Shoe& Shoe::operator=( const Shoe& shoe )
{
  *m_size = *shoe.m_size;
  *m_brand = *shoe.m_brand;

  return *this;
}

Shoe a(7, "abc");
Shoe b(4, "edf");
Shoe c(9, "ghi");

c = b = a;
```



Self-Assignment

```
class RentalSystem {
   public:
    // Assume constructor, other methods…
    RentalSystem& operator=(
        const RentalSystem & rs )
   private:
    Customer *m_customers;
    int m_nbrOfCustomers;
};

RentalSystem& RentalSystem::operator=(
        const RentalSystem & rs )
{
   delete [] m_customers;

   m_customers = new Customer[rs.m_nbrOfCustomers];
   for (int i = 0; i < rs.m_nbrOfCustomers; ++i)
    m_customers[i] = rs.m_customers[i];

   return *this;
}
```

What happens when you do the following?

```
RentalSystem r;
// Add customers…
r = r;
```

Protect from Self-Assignment

```
RentalSystem& RentalSystem::operator=(
      const RentalSystem & rs )
{
   // If this is NOT the same object as rs
   if ( this != &rs )
   {
    delete [] m_customers;

    m_customers = new Customer[rs.m_nbrOfCustomers];
    for (int i = 0; i < rs.m_nbrOfCustomers; ++i)
       m_customers[i] = rs.m_customers[i];
   }

   return *this;
}
```

Announcements

- Project 3 is out – get started now!
  - Due Thursday, March 31st
- Exam 2 is in 2 weeks
  - Will focus heavily on:
    - Classes
    - Inheritance
    - Linked Lists
    - Dynamic Memory