Today - Lecture 5 - CS202

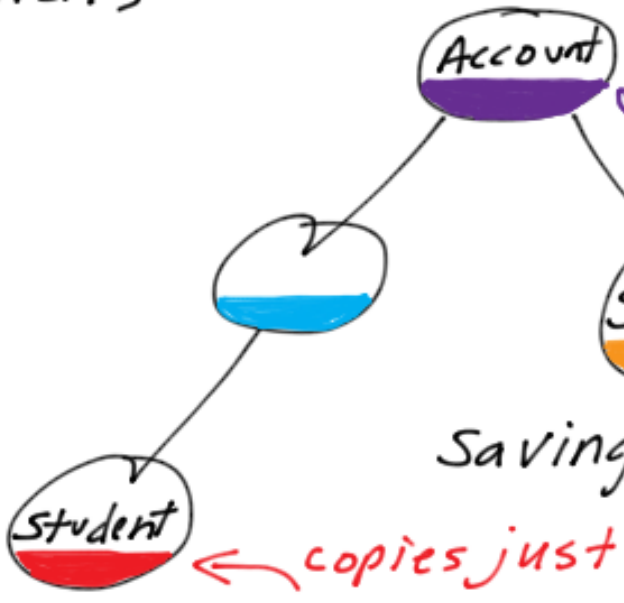
1) Operator Overloading as it relates to Inheritance (topic 3-38)
and, to dynamic binding (topic 4-35)

2) User defined Type conversion
3) Exception Handling
4) Review for the midterm

Announcements:

Inheritance and Operator Overloading

* member functions ("operators") of a derived class "hide" their parent operators as would be expected with any member functions



Account & operator=(const Account &);
(copies this)

copies this

Savings & operator=(const Savings &);

copies just this!

Student & operator=(const Student &);

Q: what are their jobs?

Q: How does the parent's data get copied?

Using base class' functionality

```
account & account::operator= (const account & source)
{
    //assume that name is a data member
    name = new char[strlen(source.name)+1];
    strcpy(name,source.name);
    return *this; //to allow for chaining
}

savings & savings::operator= (const savings & source)
{
   //First let's copy the parent's

   //Which choice is correct?
   //choice #1: *this = source;
   //choice #2: (account) *this = source;
   //choice #3: static_cast<account &> (*this) = source;
   //choice #4: account::operator=(source);

   //assume the data member if a float interest
   interest = source.interest;

   return *this;
}
```

self assignment
delete memory

not all of these are correct!

Recursion

Rvalue!!

For Members

$a + b$

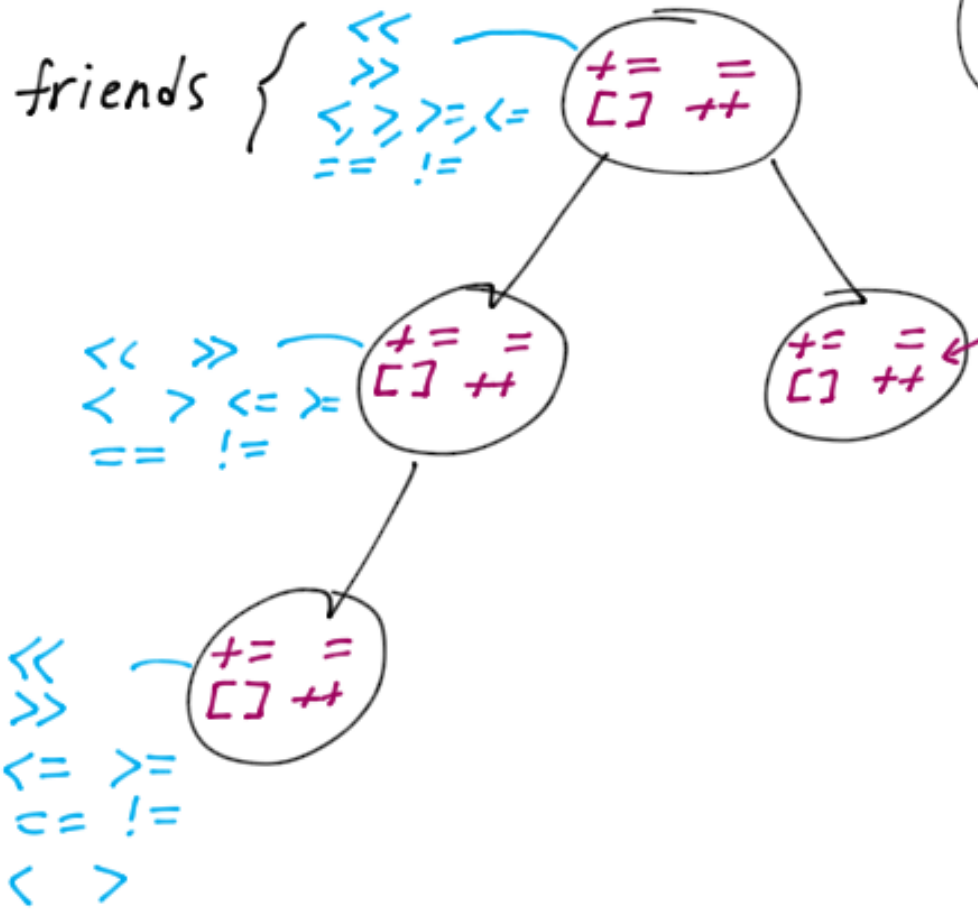
$a.operator+(b)$

$a = b$

$a.operator=(b)$

Members vs Non-Members

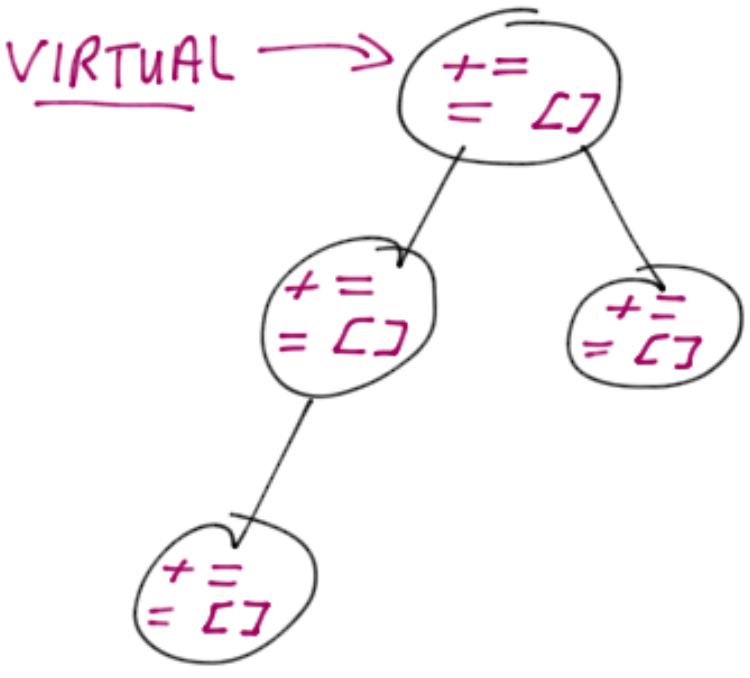
friends are NOT inherited!



Members who "hide" parent's operator. But, will explicitly call the parent's operator to perform that segment of work

(Don't have a derived class do the "job" of the base class)

With Dynamic Binding



Dynamic Binding only applies to the binding of a object TO a MEMBER FUNCTION

(It does not apply to data members or friends)

The right member operator gets invoked based on where the first operand is referencing

Virtual Member Operators

```
savings obj;
func(obj);
```

```
void func(account & base)
{
    cout << base[i];
}
```

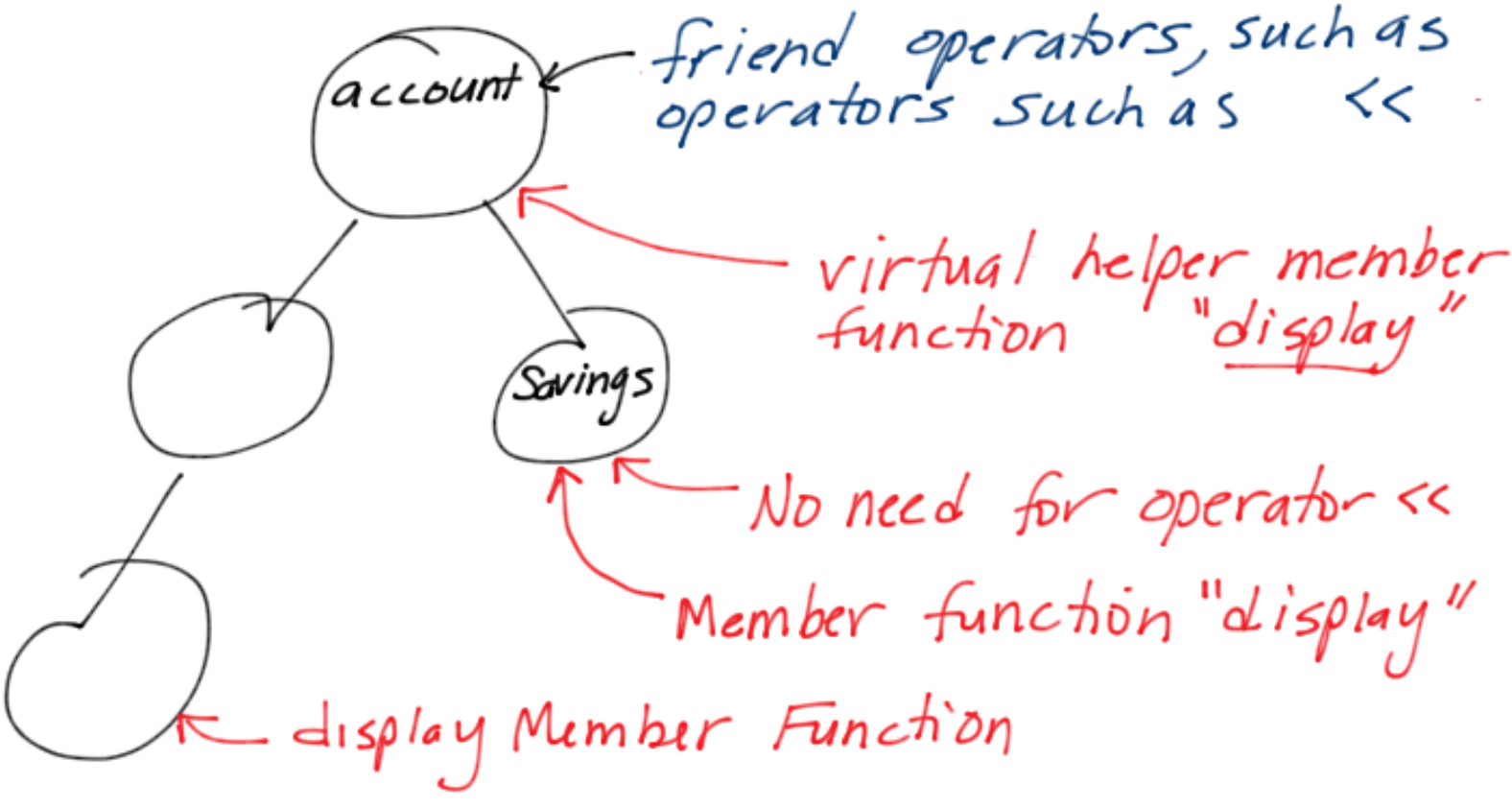
Reference to ...

```
class account
{
    public:
        account();
        virtual ~account();
        virtual account & operator = (const account &);
        virtual account & operator += (const account &);
        virtual transaction & operator[] (int index) = 0;
    private:
        char * client_name;
};

class savings: public account
{
    public:
        savings();
        ~savings();
        savings & operator= (const savings &);
        savings & operator+= (const savings &);
        transaction & operator[] (int index); //required
    private:
        transaction * list_of_transactions;
        int * num_transactions;
};
```

With Friends -

1) They cannot be virtual
2) They are not inherited
3) So, we create Virtual Member Helper functions!



account *ptr = new savings;
cout <<*ptr;
static binding

```
class account
{
    public:
        account();
        virtual ~account();
        virtual account & operator = (const account &);
        virtual account & operator += (const account &);
        virtual transaction & operator[] (int index) = 0;
        friend ostream & operator << (ostream &, const account &);
    protected:
        virtual void display (ostream &) const;
    private:
        char * client_name;
};

class savings: public account
{
    public:
        savings();
        ~savings();
        savings & operator= (const savings &);
        savings & operator+= (const savings &);
        transaction & operator[] (int index); //required
    protected:
        void display (ostream &) const;
    private:
        transaction * list_of_transactions;
        int * num_transactions;
};
```

calls:
op2->display(op1)
dynamic binding
← virtual helper member

For example

Dynamic Binding

```
ostream & operator << (ostream & out, const account & obj)
{
    obj.display(out);//calls the "RIGHT" display
                     //based on where obj references
} return out;

void account::display(ostream & out)
{
    out << name;
}

void savings::display(ostream & out)
{
    account::display(out); //display base class data
    out << any_data_members_in_savings;
}
```

Static Binding

```
account obj1;
Savings obj2;

account * ptr = &obj1;      cout << *ptr;
ptr = &obj2;                cout << *ptr;
```

Type Conversions - explicit -

"cast" `int i = (int) f;` ← floating point value

C & C#

"Functional Notation" `int i = int (f);`

C#

Function Notation only works with single names: int, float, char, class_name

So how do we represent: `ptr = (char *) name;`

✓ Casting

X Functional Notation requires a "typedef"

```
typedef char * pchar;
}
ptr = pchar (name);
```

Example of Implicit Conversions

```
class name
{
    public:
        name();
        name(char *); //allows for implicit & explicit
                      //type conversion
        name(const name &); //copy constructor
        name & operator = (const name &); // deep copy
        ~name();
    protected:
        char * a_name;
        int length;
};

//in some function....
name obj;
obj = "Sue Smith";  //causes implicit type conversion
```

Data type to convert from!

explicit

= op
copy #2

implicitly causes constructor with one arg to be called-making a deep copy

1 un-named name object

Examining the Details

```
name::name(char * a_string)
{
    length = strlen(a_string);
    a_name = new char[length +1];
    strcpy(a_name, a_string);
}

name & operator = (const name & op2)
{
    if (this == &op2) //self assignment
      return *this;
    length = op2.length;
    delete [] a_name;
    a_name = new char[length + 1];
    strcpy(a_name, op2.a_name);
    return *this;
}
```

name::

Another form of User Defined Type Conversion

```
class name
{
    public:
        name();
        name(char *);

        name(const name &);
        operator account(); //turns a name into account
        name & operator = (const name &);
        ~name();
    protected:
        char * a_name;
        int length;
};

//in some function....
account an_account;
name client_name = "Sue Smith"; //copy constructor
an_account = client_name; //but the account class only has
                          //one implementation of the = operator
                          //which is account = account
                          //causes implicit conversion
```

1) implicitly calls the operator account function

2) That makes a local copy and then returns by VALUE

3) Copy constructor is invoked upon return

4) Then, the Assignment operator is called

copies the data (4) times!

Demonstration of using type conversion

```
name::operator account()
{
   //can't have a return type

   //takes the current object and copies it into a local object
   account local;
   local.set(a_name); //a_name is a data member
                      //but we don't have access to account's
                      //data members unless we are a friend

   //time to RETURN (by value) this local:
   return local; //causes copy constructor of class
                 //account to be invoked
}
```

Pointers to Functions

```
int * ptr1;
int **ptr2;
int * ptr3[5];
int   ptr4();
int * ptr5();
int * ptr6(int *);
int (*ptr7)();
int * (*ptr8) ();


int array[5];
ptr1 = array;

*ptr1 = 10;
//or
ptr1[index] = 10;  //same as array[index] = 10;

//so, similarly
void func();
void (*ptr)();
ptr = func;

(*ptr)();
//or
ptr();  //same as func();  ...function call
```