Resource Management cont.

Copy Constructors

Copy construction is when we create (construct) a new object by copying the value of a existing object

- Used *anytime* you make a new copy of an existing class variable
    - Pass by value in function

```
void cloneANerd()
{
    PiNerd existingNerd(4); //knows PI to 4 digits
    PiNerd clonedNerd = existingNerd;
    clonedNerd.showOff(); //prints 3.141

    PiNerd clonedNerd2(existingNerd); //works the same
}
```

> ClassName(const ClassName& old)

- `const`: promise that you won't modify the old variable while constructing your new variable
- `&`: must be a reference
- type of parameter must be same type as class itself

Shallow Copy

Default C++ copy constructor: copies all the member variables from the old instance to the new instance

```
int main()
{
    PiNerd ann(3);
    if (...)
    {
        PiNerd ben = ann;
    }
    ann.showOff();
}
```

- Default constructor copies all member variables from `ann` to `ben`
    - `ben.m_n = 3`
    - `ben.m_pi` now points to the same place as `ann.m_pi`
        - points to the *original copy*
- When `ben` is destructed

  - `delete [] m_pi` is called -> deletes the original copy!
  - `ann` pointer now points to empty memory -> dangling pointer

Define your own copy constructor (**Deep Copy**)

1. Determine how much memory is allocated by the old variable
2. Allocate the same amount of memory in the new variable
3. Copy the contents of the old variable to the new variable

```
PiNerd(const PiNerd &src)
{
    m_n = src.m_n;

    //create a new array
    m_pi = new int[m_n];

    //copy each element in the array
    for (int j = 0; j < m_n; j++)
    {
        m_pi[j] = src.m_pi[j];
    }
}
```

Further Example

```
String(const String& other)
{
    m_len = other.m_len;
    m_text = new char[m_len + 1]; //neown array of characters
    strcpy(m_text, other.m_text);
}

//in f(s) we have a this pointer parameter t
    //when s is copied, our constructis called
    //1. m_len copied, t now has m_len:5
    //2. empty array created, m_text t points to this array
    //3. strcpy 'other' s to t, nm_text of t points to copy of array
```

- `other.m_len` is a private member `other`, is that allowed? YES
  - We are still in this class
  - A member function of String ctalk about the private members any String, not just of `this`
    - If we go through the publinterface, implementation mignot be as efficient
- What if this constructor usesnonconstant reference parameter?

```
String(String& other);
```

  - Copy constructor is allowed modified the "original"

Scenario: We have a data member thkeep tracks of how many times tstring has been copied

```
private:
    char* m_text;
    int m_len;
    im_numberOfCopiesMakeFromMe; };

//everytime a string is copieincrement the previous data, tcopy constructor
is modifying tobject itself
```

  - Rather uncommon but OK

- What if our constructor takes argumepassing by value? ILLEGAL

```
String (String other);
```

  - How is `other` going to initialias a copy of `s`? We have to cathe copy constructor
    - Infinite `other` s
  - The copy constructor defines whit means to pass by value, if tcopy constructor itself passes value --> infinite cycle

---

Assignment Operators

Correctly change an *existing variable*'s value to another *existing variable*

```
void reassignJoeToJan()
{
    PiNerd joe(3), jan(5);
    joe = jan;
}

joe = jan; --> joe.operator=(jan);
```

Default assignment operator: copies each data members

```
class PiNerd
{
    public:
        PiNerd(int n)
        {
            m_n =n;
            m_pi = new int[n];
            ...
        }
};

int main()
{
    PiNerd ann(3);
    PiNerd ben(4);
    ben = ann;
}
```

- Built-in assignment operator does a **shallow copy** from `ann` to `ben`
- `ben.m_n = ann.m_n`
- `ben.m_pi` pointer is assigned address stored in `ann.m_pi`
    - Neither `ann` nor `ben` now point to `ben`'s array
- Destruction process:
    - First `ben`'s destructor is called
        - Destory `ann`'s array and stuff
    - Then `ann`'s destructor is called
        - Nothing to destory
    - MEMORY LEAK: `ben`'s array was never deleted

Define your own assignment operator: `operator=`

1. Free any memory currently held by the target variable (`ben`)
2. Determine how much memory is used by the source variable (`ann`)
3. Allocate the same amount of memory in the target variable
4. Copy the contents of the source variable to the target variable
5. Return a reference to the target variable
    - Function return type is a **reference to the class**
    - returns `*this` when it's done
        - so that there's always a variable on the right hand side of the `=` for the next assignment
        - for *multiple assignments*

```
int main()
{
    Gassy sam(5, false);
    Gassy ted(10, false);
    Gassy time(2, true);

    tim = ted = sam;
}
```

  - all assignment is performed right-to-left
    - first call `ted`'s assignment operator to assign him to `sam`
    - `this` is a pointer variable that holds the **address** of the current object (`ted`'s address in RAM)
    - `*this` refers to the whole `ted` variable
  - this line returns the variable itself
    - `ted = sam` is just replaced by the `ted` variable

```
tim = ted;
```

  - then returns the `tim` variable

- `ClassName &operator=(const ClassName &rhs)`
- `const` keyword: guarantees that the `rhs` object is not modified during the copy
- MUST pass a reference to the `rhs` object

Preliminary Implementation

```
PiNerd &operator=(const PiNerd &rhs)
{
    delete [] m_pi; //free lhs object's memory
    m_n = rhs.m_n;

    //add a statement to allocate enough storage
    m_pi = new int[m_n]; //hey OS, could you reserve 12 bytes for me?

    for (int j = 0; j < m_n; j++)
    {
        m_pi[j] = rhs.m_pi[j];
    }

    return *this;
}
```

- Works properly, EXCEPT when aliasing

> Aliasing: when we use two different references/pointers to refer to the same variable

- `ann = ann;`
    - First `delete [] m_pi`: free dynamically allocated array
    - Now we have nothing to assign to -> we deleted our array!
        - We copy the random values over themselves!

> The assignment operator function must check to see if a variable is being assigned to itself, and if so, do nothing...

Copy Swap Assignment Operator

- Potential problem: dynamic allocation (`new char[m_len + 1]`) might fail due to not enough memory or other reasons
- We want our function to leavthvariables unchanged if that haenand the function needs bexecuted again

```
String u("Wow");
u = s;
//enter the assignment process//
```

- The assignment process:
    - delete object that `u`'s `m_text` is pointing to
    - assign `m_len` successfully
    - fail dynamic allocation, exception thrown, go to upper level `u=s;`
        - now `u` has a dangling pointer
        - if this line of code does not handle exception, go up to even higher level
        - local variable is destroyed, undefined behavior
    - plementation
- We don't delete old storage tiwe are sure that we got new storage
- We will need one more funio`swap`
- We create a temporary Strings copy of the rhs
    - if we can't get this storage, we leave directly, and `u` is unchanged
    - if we succeed, we could just swap value of temp and value of `u`

- Destroy local temp `` vo String::swap(String& other) { ...swap m_text and other.m_text... ...swap m_len and other.m_len... //temporary pointer and temparinteger //operations involving esbuilt-in types without dymiallocation will not throw exception }

  Stng& String::operator=(const Strinrh { if (this != &rhs) { String temp(rhs); //temp now takes value of s swap(temp); //swap temp with u, u now assigned correct value } //leave this curly bcedestructor is called for temp return *this; } `