Resource Management

Idea: If my OS has a limitation on how many files I can have open at one time --> how to manage this resource?

- There needs to be a way to release the resource after open it: open the file, use it, close the file --> then we can have at most one file open at a time

Quick Review: Pointers

We use pointers to efficiently access/modify variables defined in other parts of our program

> A pointer is a variable that holds another variable's address

```
void someFunction()                | grade 'B' 00001024
{                                  |
    char grade = 'B';              | ptr 1024  00001026
    char *ptr = &grade;            |                -
}                                  |           0001030
```

- Address of a variable:
    - The **lowest** address in memory where the variable is stored
    - `&` AddressOf operator: gives numerial address of a variable

Passing Pointer into a Function

```
void set(int *px)
{
    *px = 5;
}

int main()
{
    int x = 1;
    set(&x);
    cout << x;
}
```

- Address of `x` is passed to the function
- Pointer `px` now stores address of `x`
- Deference `px` and change `x` value to 5

Pointers vs References

- When you pass a variable by reference to a function, what really happens?

```
void set(int &val)
{
    val = 5;
}

int main()
{
    int x = 1;
    set(x);
    cout << x;
}
```

- Reference is just a simpler notation for passing by a pointer
- It looks like we are passing the value of `x`, but actually we are passing the address of `x`

> Always initialize pointers to nullptr immediately when you defined them

```
double *ptr_to_debt = nullptr;
```

Arrays, Addresses and Pointers

```
int main()
{
    int nums[3] = {10, 20, 30};
    cout << nums; //prints address of nums
    int *ptr = nums; //pointer to array
}
```

In C++, a pointer to an array can be used just as if it were an array itself

```
cout << ptr[2]; //prints nums[2] or 30
```

The dereference operator can access array elements

```
cout << *ptr; //prints nums[0]
cout << *(ptr + 2); //prints nums[2]
```

The two syntaxes have identical behavior: Get the value in `ptr`, and go to that address in memory, then skip down `j` elements and get the value

```
ptr[j]
*(ptr + j)
```

When you pass an array to a function -> passing the address to the start of the array

Classes and the "this" Pointer

```
class Wallet
{
    public:
        void Init();
        void AddBill(int amt);
    private:
        int num1s, num5s;
};

int main()
{
    Wallet a;
    a.Init();
    a.AddBill(5);
}
```

Everytime you call a member function of an object -> C++ invisbily rewrites your function call and passes in the variable's address

```
a.addBill(5); -> addBill(&a, 5);
a.Init(); -> Init(&a);

void Wallet::Init() ->

void Init(Wallet *this)
{
    this->num1s = this-> num5s = 0;
}
```

`this` is a variable that stores the address of the current object

Pointers to Functions

```
void squared(int a)
{
    cout << a*a;
}

void cubed(int a)
```

```
{
    cout << a*a*a;
}

int main()
{
    void (*f)(int); //declare a function pointer

    f = &squared; //gets the address of squared function
    f(10); //use like a regular function call

    f = &cubed; //the & operator is optional
    f(2);
}
```

Dynamic Memory Allocation

New and Delete For Arrays

```
int size, *arr;
arr = new int[size];
arr[0] = 100;
delete [] arr;
```

New and Delete in a Class

```
class PiNerd
{
    public:
        PiNerd(int n)
        {
            //Allocate array
            m_pi = new int[n];
            m_n = n; //store size

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

        void showOff()
        {
            for (int j = 0; j < m_n; j++)
            {
                cout << m_pi[j] << endl;
            }
        }
```

```
    private:
        int *m_pi; //pointer variable
        int m_n; //size variable
}
```

We need a destructor that frees dynamically allocated array

```
~PiNerd()
{
    delete [] m_pi; //free memory
}
```

Copy Constructor, Assignment Operator

Copy Construction

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];
    }
}
```

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...

Example: Memory Management, create a String Type

Scenario: there is no standard String type library, we will write a C++ string type (we do have C strings)

```
//a function to create a string
void h()
{
    String s("Hello"); //C string is an array of characters
}

//[0] [1] [2] [3] [4] [5]
//'H' 'e' 'l' 'l' 'o' '\0'

class String
{
    public:
        String(const char* value);
        String(); //default constructor for empty string
    private:
        //data member to store the array of characters
        char m_text[100];  //this is problematic!! what if longer string?
        char m_text[100000]; //waste lots of space, each string will take up
100000 bytes

        //dynamic allocate an array of characters big enough to hold a string
```

```
        char* m_text;
};
```

- Allocating storage for an empty string
    - Case 1: empty string will have no characters followed by a zero byte '\0'
        - consistent with every other string, no special cases, easier implementation

      however...

        - if there are a lot of strings, cost of creating even a 1 byte array is a lot of instructions executed, then destroying them costs more
        - space cost dynamically is more than 1 byte per empty string (storage allocator also has to have overhead bookkeeping information)
    - Case 2: or just a nullptr
        - **saves space**: no storage overhead, more practical
            - more complicated implementation to check for special case
- String length: many functions will need to know the string length to process
    - this is an **expensive operation**: walk down every byte until the last to know the length
    - therefore, we keep the length as data member in the string object (for performance reasons)
        - in the real world: you will want to measure if this is more efficient than counting each time

```
class String
{
    public:
        String(const char* value);
        String();
        ...
    private:
        //Class invariant:
            //m_text is a pointer to a dynamically allocated array of m_len
+ 1 characters
            //m_len >= 0
            //m_len == strlen(m_text)
        char* m_text;
        int m_len;
}
```

- **Constructor implementation**

```
//WRONG
String::String(const char* value)
{
    m_text = value; //value is a pointer to constant characters, but m_text
is a pointer to something that is not const
    m_len = strlen(value);
}

char buffer[1000]; //character array
cin.getline(buffer, 1000); //get line of input, user types hello
//buffer has "Hello\0"//
String s(buffer);
//m_text has value, points towards text in buffer, length is 5

//now confusing: if user inputs more
cin.getline(buffer, 1000);
//now buffer: wow\0he
//now the buffer has changed, now S's value is not hello anymore, the length
changed as well
```

Fixing this issue...

```
String::String(const char* value)
{
    m_len = strlen(value);
    m_text = new char[m_len + 1]; //plus zero byte
    strcpy(m_text, value); //copy from value into m_text (all arrays)
}

/* now
s: m_text: h e l l o '\0'
m_len: 5
*/

//Default constructor
String::String()
{
    m_len = 0;
    m_text = new char[1];
    m_text[0] = '\0'
}

/* now
s: m_text: '\0'
m_len: 0
*/
```

Refactoring constructor code

```
//Manipulate the two to be extremely similar
String::String()
{
    m_len = strlen("");
    m_text = new char[m_len + 1];
    strcpy(m_text, "");
}

//Then we don't need a default constructor, just give a default value to the
previous constructor

String(const char* value = "");
```

Check for nullpointer (if `value` is a nullptr)

- the null pointer should get an empty string as well

```
String::String(const char* value)
{
    if (value == nullptr)
    {
        value = ""; //compiler set up an array of one character with a zero
byte in memory somewhere
    }
    //value is a pointer to const characters, you cannot modify those const
characters through 'value', but you can make value point to other stuff

    m_len = strlen(value);
    m_text = new char[m_len + 1];
    strcpy(m_text, value);
}
```

NOW WE HAVE MEMORY LEAKS!!!!!!!!

```
void h()
{
    String s("Hello"); //no destructor
}
```

- Destructors
  - Steps of destruction:
    1. Execute the body of the destructor
    2. Destory each data member:
       - If built-in type, do nothing -If class type, call that class's destructor for that member

3. ...

```
//WRONG!!!!!!
String::~String()
{
    delete m_text;
}

Blah* bp = new Blah;  //create one blah object, bp point to it
...
delete bp;

Foo* fp = new Foo[100];
...
delete [] fp; //must use array form of delete
```

- Deletion differs for regular versus array objects
    - has to do with different forms of bookkeeping information

```
String::~String()
{
    delete [] m_text;
}
```

- Copy Constructor
    - Scenarios:

    ```
    String x(s);
    String x = s;
    //Pass by Value
    //Returning a value from a function (not a pointer or reference)
    String blah (..., ..., ...)
    {
        String result;
        ...
        return result;
    }
    ```

    - Example: Passing String Type by Value into a Function

    ```
    void f(String t)
    {
        ...
    }
    ```

```
void h()
{
    String s("Hello"); //successfully created String object
    f(s);
}
```

- We are passing by value here, String will be copied
- Default (if we did not specify how String should be copied)
  - Copy each member: copy pointer (new pointer that points to the same hello, copy value length = 5)
  - Weird: if function modifies string
    - "hello" --> "jello"
    - The original object String s has been modified as well (since we changed "hello" through the pointer)
  - Even more problematic: when the function `f` return`, local variables are destroyed
    - `t` is going to go away
    - destructor is called on `t`, which is told to delete the array that this pointer points to
    - Then the array is GONE!!!!
    - `s` is now a dangling pointer --> **undefined behavior** if you try to following the pointer `s`
    - When we leave the function `h`: the destructor is called
      - Destructor is going to delete something that already has been deleted --> **undefined behavior** --> crash
- We should make the copy of the original to be **completely** independent
- **Copy Constructor**: Declare an appropriate way to create a String from another String
  - Called whenever a string is created from another string

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

//in f(s) we have a this pointer to parameter t
```

```
    //when s is copied, our constructor is called
    //1. m_len copied, t now has m_len:5
    //2. empty array created, m_text of t points to this array
    //3. strcpy 'other' s to t, now m_text of t points to copy of
array
```

- `other.m_len` is a private member of `other`, is that allowed? YES
    - We are still in this class
    - A member function of String can talk about the private members of any String, not just of `this`
        - If we go through the public interface, implementation might not be as efficient
- What if this constructor uses a nonconstant reference parameter?

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

    - Copy constructor is allowed to modified the "original"

  Scenario: We have a data member that keep tracks of how many times the string has been copied

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

  //everytime a string is copied, increment the previous data,
  the copy constructor is modifying the object itself
  ```

    - Rather uncommon but OK
- What if our constructor takes argument passing by value? ILLEGAL

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

    - How is `other` going to initialize as a copy of `s`? We have to call the copy constructor
        - Infinite `other` s
    - The copy constructor defines what it means to pass by value, if the copy constructor itself passes by value --> infinite cycle

- Assignment
    - Take an already existing string and giving it a different value
        - This is different from creating a brand new string as a copy of another string
            - Brand new string storage has nothing in it
            - If assignment, string already has an old value
            - Starting condition is different

```
s = t; //assignment
String x(s); //copy construction

String y = s; //what is this?
//this is COPY CONSTRUCTION, NOT ASSIGNMENT
//we are creating a brand new object that did not exist before
```

Example:

```
void h()
{
    String s("Hello");
    String u("Wow");
    u = s;
}
```

- Default: if we don't define an assignment operator
    - Compiler will give a default assignment operator that assigns each data member to each
    - What happens:
        - We created `u`:
            - m_text -> "Wow\0"
            - m_len = 3
        - Compiler assigns each data member `u = s;`
        - Assigns pointer to "Hello\0', m_len = 5;
            - SAME PROBLEM: `u` and `s` m_text are pointing to the same piece of data
                - When destroyed, there will be dangling pointer and weird changes
            - Problem 2: Memory Leak

        - "Wow\0" never gets destroyed since `u`'s `m_text` is pointing somewhere else, and that gets destroyed, "Wow\0" is left somewhere in memory storage
    - We can do default if all data members are simple types
- Assignment will be a member function of the class: write your own version of the assignment operator
    - The operator `=` is just shorthand for `u.operator=(s);` //call a function
    - Traditional approach
        - The this pointer points to `u` as it is the object being modified
        - We pass `s` by constant reference
        - We assign over the length
        - Then we delete old storage (to prevent memory leak)
        - Then dynamically assign, copy over text

    ```
    void String::operator=(const String& rhs) //NOT QUITE CONVENTIONAL
    {
        delete [] m_text; //this->m_text, rhs is a reference to s
        m_len = rhs.m_len; //NOT QUITE RIGHT
        m_text = new char[m_len + 1];
        strcpy(m_text, rhs.m_text);
    }
    ```

    Add the convention: make assignment operator usable like the way for built-in types

    - for built-in types, assignment returns a value

        ```
        int k = 3;
        int n = 5;
        int m;
        m = (k = n); //n assigned to k, result of this assignment assigned to m
        ```

    - for Strings, we want the same effect...

        ```
        v = (u = s);
        ```

    - when you overload an operator, return the new value of the left hand side

    ```
    String String::operator=(const String&rhs)
    {
        delete [] m_text; //this->m_text, rhs is a reference to s
    ```

```
    m_len = rhs.m_len; //NOT QUITE RIGHT
    m_text = new char[m_len + 1];
    strcpy(m_text, rhs.m_text);
    //return this; NO!!!, this is a pointer to a string, we want
to return the string itself

    return *this;
}
```

- Problem: ^ this is INEFFICIENT
  - We are returning a value --> this will be a copy of return expression
  - Take `*this` the left hand side, and make copy to return to call
    - `v = (u = s)`: v.operator=(u.operator=(s))
      - result of the first call is a *copy* of `u`
      - this result will be used as argument to second call
      - WHY NOT JUST LET `v` assign from `u` directly?
        - how to make function return not a copy but another name for `u`?
- Convention: assignment operator returns reference to object

```
String& operator=(const String& rhs);
```

Fix the "NOT QUITE RIGHT" problem: aliasing

- Scenario: we have an array of Strings
  - Strings get assigned around from one element to another with an algorithm, every so often, Strings have same value --> so we assign a String to itself
    - For simple types, if you do self assignment, value **unchanged**

    ```
    int k = 3;
    k = k;
    ```

    - Should have **no effect**
  - With our current implementation

  ```
  this -> u: m_text -> "Wow\0"
             m_len = 3
  ```

```
u = u;
u.operator=(u);
//now our pointer and reference both indicate the same
object
```

- First, we delete old value, call delete object pointed to by m_text
- but now, we've lost the "Wow\0"!
- Then, we set `m_len = rhs.m_len;` this is harmless
- Then we set a brand new array of characters, uninitialized
- Then we did string copy `strcpy(m_text. rhs.m_text);`
  - We are string copying this array into itself -> this is **undefined behavior**, it will copy byte to itself and stop if it is a zero byte
  - May not crash, might return -> but still have undefined behavior, lost value

- Fix: Test!
  - If the lhs object is the same as the rhs object, then just return immediately
  - How to compare?

```
//WRONG
if (this == rhs) //type is different, this is pointer,
rhs references a string
```

- Look at their **address**

```
if (this == &rhs)
```

- We could also look at their **value** but we have to define a not equal operator for String comparison, but this is comparing if the two strings have the same value

```
*this != rhs
```

  - Advantage: skips cost to copy if value same but different object
  - However: cost to check if string is same value, perhaps it is not worth the check

- The classic solution:

- Not the modern way to do it, but fixes the problem

```
String String::operator=(const String&rhs)
{
    if (this != &rhs)
    {
        delete [] m_text;
        m_len = rhs.m_len;
        m_text = new char[m_len + 1];
        strcpy(m_text, rhs.m_text);
        return *this;
    }
}
```

- Some notes: simplifying the assignment function
    - We can't call destructor and copy constructor easily inside the assignment operator function
    - We COULD write helper functions that get rid of repeated code and have all three functions call those helper functions, this is the classic way...
- Modern Approach
    - Potential problem: dynamic allocation (`new char[m_len + 1]`) might fail due to not enough memory or other reasons
        - We want our function to leave the variables unchanged if that happens and the function needs to be executed 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
    - Implementation

- We don't delete old storage until we are sure that we got new storage
- We will need one more function `swap`
- We create a temporary String as a 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

```
void String::swap(String& other)
{
    ...swap m_text and other.m_text...
    ...swap m_len and other.m_len...
    //temporary pointer and temporary integer
    //operations involving these built-in types without dynamic allocation will not throw exception
}

String& String::operator=(const String&rhs)
{
    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 brace, destructor is called for temp
    return *this;
}
```

Aside: Aliasing

- Two pointers or references to the same type
- Previously, if assigning a String to itself
- Example:

```
void transfer(Account& from, Account& to, double amt)
{
    if (amt > from.balance())
    {
        ...error message...
    }
    else
    {
        from.debit(amt);
        to.credit(amt);
```

```
    }
}
```

- What if `from` and `to` are referencing the same account??

  ```
  transfer(aa[i], aa[j], 20000);
  ```

  - we are transfering 20000 from an account to itself
  - it might seem harmless, but there are no scenarios
    - what if balance is only 5000, error message show up, but you're not actually transferring money...
    - what if there is a debit fee when you debit money, etc.. if you're not really transferring money, then should you still be charged this fee?

```
if (&from != &to)
```