

# 601.220 Intermediate Programming

Summer 2024, Meeting 18 (July 12th)

# Today's agenda

- Review exercises 27 and 28
- Day 29 recap questions
- Exercise 29
- Day 30 recap questions
- Exercise 30

→ operator overloading  
→ copy constructor and  
assignment operator.

# Reminders/Announcements

CTrie  $\rightarrow$  chars  
TTrie  $\rightarrow$  template v

- HW7 is due **Thursday, July 18st**
  - We will have covered everything you need to know for the CTrie class as of today
  - The TTrie class is a template class: we will cover template classes and functions on Monday
- Final project team formation
  - Submit the **Google form in Piazza** (pinned) by **12:15 pm on Monday (July 15th)**  $\rightarrow$  post 66 in piazza.
  - If you aren't registered as being on a team by the deadline above you will be assigned to a team.

## Exercise 27 review

### Part 2: mean and median functions

GradeList  
→ grades (Vector<double>)

```
double GradeList::mean() {  
    assert(!grades.empty());  
    double sum = 0.0;  
    for (std::vector<double>::const_iterator i = grades.cbegin();  
         i != grades.cend();  
         ++i) {  
        sum += *i;  
    }  
    return sum / grades.size();  
}
```

```
double GradeList::median() {  
    return percentile(50.0);  
}
```

## Exercise 27 review

UML

GradeList

— grades (Vector<double>)

+

Part 3: in main2.cpp:

```
GradeList gl;
```

```
double min_so_far = 100.0;
for (size_t i = 0; i < gl.grades.size(); i++) {
    if (gl.grades[i] < min_so_far) {
        min_so_far = gl.grades[i];
    }
}
```

This does not work because grades is a private member of GradeList, so a main function (which is not a member of GradeList) cannot access it directly.

## Exercise 27 review

encapsulation

Part 3: one possible solution is

```
// in grade_list.h (adding new public member functions)
```

```
size_t get_num_grades() const { return grades.size(); }  
double get_grade(size_t i) const { return grades[i]; }
```

```
// in main2.cpp
```

```
double min_so_far = 100.0;  
for (size_t i = 0; i < gl.get_num_grades(); i++) {  
    if (gl.get_grade(i) < min_so_far) {  
        min_so_far = gl.get_grade(i);  
    }  
}
```

## Exercise 27 review

Another possible solution: add to `grade_list.h` (in `GradeList` class)

```
const std::vector<double> &get_grades() const  
{ return grades; }
```

In `main2.cpp`, change `g1.grades` to `g1.get_grades()`.

Arguably, this doesn't violate encapsulation because a `const` reference can't be used to modify the internal data of the `GradeList` object. However, it does result in "leaking" the knowledge that the grades in a `GradeList` are stored in a `std::vector<double>`.

## Exercise 27 review

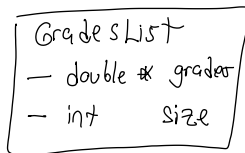
```
// Part 4 (main3.cpp)
#include <iostream>
#include "grade_list.h"

int main() {
    GradeList gl;
    for (int i = 0; i <= 100; i += 2) {
        gl.add(double(i));
    }
    std::cout << "Minimum: " << gl.percentile(0.0) << std::endl;
    std::cout << "Maximum: " << gl.percentile(100.0) << std::endl;
    std::cout << "Median: " << gl.median() << std::endl;
    std::cout << "Mean: " << gl.mean() << std::endl;
    std::cout << "75th percentile: " << gl.percentile(75.0) << std::endl;
}
```



## Exercise 28 review

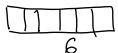
UML



*// GradeList constructor*

```
GradeList::GradeList(int capacity)
    : grades(new double[capacity])
    , capacity(capacity)
    , count(0) {
}
```

## Exercise 28 review

  $\leadsto 12 \leadsto 24 \leadsto 48$   
 $1,000,000 \leadsto 2,000,000$

*// GradeList add member function*

```
void GradeList::add(double grade) {  
    if (count >= capacity) {  
        double *expanded = new double[capacity * 2];  
        for (int i = 0; i < count; i++) {  
            expanded[i] = grades[i];  
        }  
        delete[] grades;  
        grades = expanded;  
        capacity *= 2;  
    }  
    grades[count++] = grade;  
}
```

Managing  
the capacity

## Exercise 28 review

```
// GradeList add (many) function
void GradeList::add(int howmany, double *grades) {
    for (int i = 0; i < howmany; i++) {
        add(grades[i]);
    }
}

// GradeList clear function
void GradeList::clear() {
    delete[] grades;
    grades = new double[1];
    capacity = 1;
    count = 0;
}
```

## Exercise 28 review

Memory leak reported by valgrind:

```
==4874==
==4874== HEAP SUMMARY:
==4874==      in use at exit: 64 bytes in 1 blocks
==4874==    total heap usage: 9 allocs, 8 frees, 74,016 bytes allocated
==4874==
==4874== LEAK SUMMARY:
==4874==    definitely lost: 64 bytes in 1 blocks
==4874==    indirectly lost: 0 bytes in 0 blocks
==4874==    possibly lost: 0 bytes in 0 blocks
==4874==    still reachable: 0 bytes in 0 blocks
==4874==         suppressed: 0 bytes in 0 blocks
==4874== Rerun with --leak-check=full to see details of leaked memory
```

## Exercise 28 review

Adding a destructor:

```
// in grade_list.h (in the GradeList class definition)
```

```
~GradeList();
```

```
// in grade_list.cpp
```

```
GradeList::~GradeList() {  
    delete[] grades;  
}
```

## Exercise 28 review

main2.cpp requires a default constructor:

```
// in grade_list.h
```

```
GradeList();
```

GradeList e;

```
// in grade_list.cpp
```

```
GradeList::GradeList()  
: grades(new double[1])  
  , capacity(1)  
  , count(0) {  
}
```

## Exercise 28 review

STL

*// begin() and end() functions can be defined in grade\_list.h*

```
double *begin() { return grades; }  
double *end()   { return grades + count; }
```

Pointers can be used as iterators because they support the essential operations (dereference, advance using ++, == and != to compare) required for iterator values.

## Day 29 recap questions

- ① What is overloading in C++?
- ② Can you overload a function with the same name, same parameters, but different return type?
- ③ Is it true that we can overload all the operators of a class?
- ④ What is a copy constructor? When will it be called?
- ⑤ What happens if you don't define a copy constructor?
- ⑥ What is the `friend` keyword? When do we use it?



# 1. What is **overloading** in C++?

function

*Overloading* means define two or more functions (or member functions) with the same name.

This is allowed as long as the overloaded variables can be distinguished by number and/or types of parameters, or by **const-ness**.

Note that you used overloading in exercise 28: there were two add member functions in the GradeList class.

```
void add (double g)
```

```
void add (double* grades, int how-many)
```

2. Can you overload a function with the same name, same parameters, but different return type?

No. Overloaded variants must be distinguishable by their argument(s) or constness.

### 3. Is it true that we can **overload all the operators** of a class?

Mostly. You can't overload the "." (member selection) or "::" (scope resolution) operators. All other operators may be overloaded.

MyClass a, b;

a + b;

a << b;

\*  $\leadsto$  ① initialize ptr ② dereference a ptr.

## 4. What is a copy constructor? When will it be called?

A copy constructor initializes an object by copying data from another object of the same type.

E.g.:

`String (String&other)`

```
std::string s("hello");
```

```
std::string s2(s);    // initialized using std::string's copy ctor  
std::string s3 = s;  // initialized using std::string's copy ctor
```

The copy constructor is called any time an instance of a class needs to be initialized by copying an object of the same type. This includes passing an object to a function by value, and (maybe!) when returning an object from a function by value.

(It's possible for the compiler to use "return value optimization" so that an object returned by value is constructed in the caller's stack frame, without the need for copying.)

## 5. What happens if you don't define a copy constructor?

The compiler will generate a copy constructor automatically if one isn't explicitly defined.

The compiler-generated copy constructor will copy field values in order. (This is known as “member-wise” copying.)

Note: if the class has a non-trivial destructor (e.g., the destructor deallocates dynamic memory), member-wise copying in the copy constructor will result in serious program bugs. We'll discuss this in a bit.

# Notes

## Copy constructor

- ① String s ("another str");
- ② String s = "another str";
- ③ Passing custom classes by value.

```
void my_func ( MyStr s ) { }
```

```
MyStr s ( s1 );
```

automatically

```
main ()  
{  
    MyStr s1;  
    my_func (s1);  
}
```

## 6. What is the **friend** keyword? When do we use it?

```
MyClass  
- int a;
```

```
void my_func ( MyClass a )  
{  
    a.a;  
}
```

The **friend** keyword allows a **non-member function** to be granted access to private members of a class.

It's *occasionally* useful for things like stream insertion and extraction (<< and >>), which can't be class members, but may need to access the internal data representation of an object.

## Exercise 29

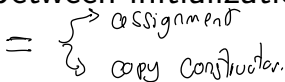
- `Complex` class to represent complex numbers
- Overload operators to do arithmetic
- Overloaded stream insertion operator (`<<`), as a friend function
- Breakout rooms 1–10 are “social”
- Use Slack to let us know if you have questions!



## Day 30 recap questions

- ❶ What is difference between initialization and assignment?
- ❷ Does the line `f2 = f1;` use initialization or assignment (assume `Foo` is a class and `f1` and `f2` are both of type `Foo`)?
- ❸ Does the line `Foo f2 = f1;` use initialization or assignment (assume `Foo` is a class and `f1` is of type `Foo`)?
- ❹ What is a shallow copy and what is a deep copy?
- ❺ What is the rule of 3?

# 1. What is difference between initialization and assignment?

= 

Initialization: a constructor is called when an object's lifetime begins.

Assignment: the = operator (assignment) is used to assign new data to an existing object, replacing its current contents.

Examples:

```
std::string s("hello");           // initialization of s

std::string s2 = "hello again";  // initialization of s2
std::string s3;                  // initialization of s3
                                // using default ctor
s3 = s;                          // assignment to s3
```

2. Does the line `f2 = f1;` use initialization or assignment (assume `Foo` is a class and `f1` and `f2` are both of type `Foo`)?

Assignment. It is not a variable declaration of `f2`, so `f2` has already been initialized.

3. Does the line `Foo f2 = f1;` use initialization or assignment (assume `Foo` is a class and `f1` is of type `Foo`)?

Initialization. This is a variable declaration of `f2`, and `f1` is being provided as the initial value, so the copy constructor is called to initialize `f2` with `f1`'s contents.

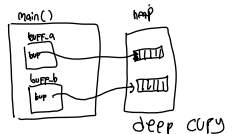
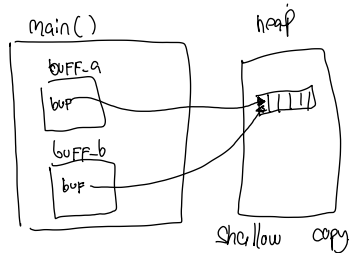
## 4. What is a shallow copy and what is a deep copy?

Deep copy: replicate dynamically-allocated objects/arrays.

Shallow copy: just copy pointers to dynamically-allocated objects/arrays.

Example class:

```
class CBuf {  
private:  
    char *buf; int capacity;  
public:  
    CBuf(int capacity)  
        : buf(new char[capacity]), capacity(capacity) { }  
    CBuf(const CBuf &other); // copy ctor  
    // ...other member functions...  
};
```



## Copy constructor using deep copy

```
CBuf::CBuf(const CBuf &other)
: buf(new char[other.capacity])
, capacity(other.capacity) {
    for (int i = 0; i < capacity; i++) {
        buf[i] = other.buf[i];
    }
}
```

The new object will have its own dynamically-allocated array, distinct from the original object.

## Copy constructor using shallow copy

```
CBuf::CBuf(const CBuf &other)
    : buf(other.buf), capacity(other.capacity) {
}
```

Shallow copy means two objects have pointers to the same dynamically-allocated array. If one object modifies the array, the changes are visible in the other object (because they are “sharing” the array.) Also: which object’s constructor should delete it?

# Shallow copy vs. deep copy

Shallow copy tends to be problematic because either

- Multiple objects try to deallocate the dynamic memory (double free, a serious memory error)
- No object tries to deallocate the dynamic memory (memory leak, also a fairly serious bug)

If you are implementing a class that manages dynamic memory, the copy constructor and assignment operator should do deep copy.



## 5. What is the rule of 3?

If a class has a non-trivial destructor (e.g., the destructor deletes a dynamically-allocated object or array), then it also needs

- a copy constructor
- an assignment operator

Both of these should do a *deep copy*.

# Disabling value semantics

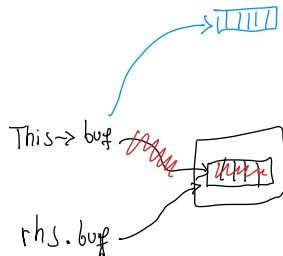
Alternately: a class with a nontrivial destructor could *prohibit* the copy constructor and assignment operator from being used by defining them in the class definition as `private`, and then not defining them.

The disadvantage of this approach is that the class will not have *value semantics*. So you can't copy, assign, pass by value, return by value, etc.

# A subtle issue with assignment

Consider the following assignment operator:

```
CBuf &CBuf::operator=(const CBuf &rhs) {  
    delete[] buf;  
    buf = new char[rhs.capacity];  
    capacity = rhs.capacity;  
    for (int i = 0; i < capacity; i++) {  
        buf[i] = rhs.buf[i];  
    }  
    return *this;  
}
```



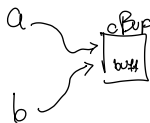
Can you spot the problem? (It is very subtle.)

# Guarding against self-assignment

We can't rule out the possibility that an object might be assigned to itself!

While this (probably) wouldn't happen explicitly, it could happen fairly easily because of references:

```
void foo(CBuf &a, CBuf &b) {  
    if ( /* some condition */ ) {  
        a = b; // do we really know that a and b  
               // refer to different objects?  
    }  
}
```



## Buggy version of assignment operator

```
CBuf &CBuf::operator=(const CBuf &rhs) {  
    delete[] buf;  
    buf = new char[rhs.capacity];  
    capacity = rhs.capacity;  
    for (int i = 0; i < capacity; i++) {  
        buf[i] = rhs.buf[i];  
    }  
    return *this;  
}
```

Think about what happens when `rhs` and `*this` are the same object. The character array is deleted, but then we try to copy data from the uninitialized newly-allocated character array.

## Fixing the bug

```
CBuf &CBuf::operator=(const CBuf &rhs) {  
    if (this != &rhs) {  
        delete[] buf;  
        buf = new char[rhs.capacity];  
        capacity = rhs.capacity;  
        for (int i = 0; i < capacity; i++) {  
            buf[i] = rhs.buf[i];  
        }  
    }  
    return *this;  
}
```

Now the assignment properly does nothing if `rhs` and `*this` are the same object.

You should get into the habit of using this idiom when you implement assignment operators.

## Exercise 30

- Linked list implementation in C++
- Implementation of copy constructor and assignment operator using deep copy
- Breakout rooms 1–10 are “social”
- Use Slack to let us know if you have a question!

# Notes



# Notes

# Notes

# Notes