COSC 2P95 – Lab 5 – Section 01/02 – Object Orientation

For this lab, we'll be looking at Object Orientation in C++.
This is just a cursory introduction; it's assumed that you both understand the lecture examples, and will be using the reference of your preference to find additional information.

Step One: Declaring Classes

We're going to start simple: a basic template for a trivial record with public accessors and updaters. These member functions will operate on the two member variables.

```
class Record {
private:
	long rid; //Record ID
	char *content;
public:
	long getRID() {
		return rid;
	}

	char *getContent() {
		return content;
	}

	void setRID(long rid) {
		this->rid=rid;
	}

	void setContent(char cstring) {
		//How would we do this? Do we allocate? Beware memory leaks!
	}
};
```

Note the `this` pointer. This is an implicit pointer provided to *methods* (i.e. those member functions performed by the object).

Step Two: Constructors

Of course, what we're missing is a way to easily initialize the object. The updaters aren't really an ideal substitute. For this example, we'll actually phase out those updaters. Let's start with a single constructor: we have two member variables (instance variables), so we'll take two parameters.

```
Record(long rid, char content[]) {
	int clength=0;
	this->rid=rid;
	while (content[clength]!='\0') clength++;
	this->content=new char[clength];
	for (int i=0;i<clength;i++) this->content[i]=content[i];
}
```

Using this record class is pretty easy.
e.g.

```
Record r(23L, “contented”);
char c[80];
cin>>c;
Record s(45L,c);
```

Well, that's not bad, but we have a problem: we're dynamically allocating memory on the heap, but what happens when we no longer need the record? The pointer can be easily reclaimed, but not the memory it's pointing to. Though we could hypothetically add a method to release the memory when manually invoked, that's not a reliable approach. Instead, we want to guarantee that some code will run when the object is deallocated. i.e. we need a destructor.

Step Three: Destructor

Simply add the following:

```
~Record() {
      delete[] content;
}
```

(okay, so maybe this didn't need to be its own step)

So now we're done, right? … yeah… sure…

Query: remembering that classes are based on structs, what happens if we pass a Record to a procedure to modify? Will it change the original or not?

Step Four: Copy Constructor

Temporarily make the following changes:

- Comment out your destructor
- Make the instance variables public
- Add the following procedure to the file (i.e. **not** to the class)

```
void mutate(Record r) {
     r.rid=77;
     r.content[0]='Z';
}
```

Suppose you invoked it with:

```
Record r(23,"howdy there!");
cout<<r.rid<<":"<<r.content<<endl;
mutate(r);
cout<<r.rid<<":"<<r.content<<endl;
```

Before you actually run it, what would you expect?
Now try running it.

Ouch! What happened?

It's true. Just like structs, when you use an object as an argument to a procedure, it wants to create a copy. It passes by value. However, one of these values is the pointer `content`, which contains a fixed place in memory.
Basically, if the original record contained an rid of 23 and a content address of 0xFF08, then the new (copy) record will have a different long that happens to contain 23, and a different content address that happens to hold the value 0xFF08.

So, why did we comment out the destructor? Because, if it isn't commented, then when we leave the procedure it automatically deallocates the parameter, which includes a deallocation of 'content' from the heap. Once the program ends, it also tries to deallocate, including deleting the 'content' (which has already been deleted).

The solution? *Overload* the copy constructor, so that we can make the pointer contain a different memory address.

Add the following:

```
Record(const Record &original):rid(original.rid) {
      int clength=0;
      while (original.content[clength]!='\0') clength++;
      clength++;
      this->content=new char[clength];
      for (int i=0;i<clength;i++) this->content[i]=original.content[i];
}
```

Also, uncomment the destructor, and change the instance variables to being private again.

Step Five: Stream Insertion (Output)

When it comes to output, we *could* simply use the accessors to get the two member variables, and we *could* even write a simple procedure to accept a reference to the record and print out the record. What's more, if we also had another parameter for the specific ostream, then we could use it for both terminal and file output.

But what operation would that actually rely on? The stream insertion (`<<`). Why can't we just define a stream insertion operator for our record?
Naturally, we can.

Overloading operators is relatively simple. The biggest catch is that operators defined as member functions must use that object as the left-hand side operand. But stream insertion uses the ostream as the left-hand side. As such, it can't be a normal member function (i.e. it can't be a proper method).
However, if we write a separate function, it will still need access to the inner contents of the record. To achieve all of this, we simply use a *friend function*.

First, to the class, let's add a friend declaration:

```
friend std::ostream& operator<<(std::ostream &out, Record &rec);
```

Make sure to put it under the public area of the class.

Now, somewhere else within the class, let's add the implementation:

```
std::ostream& operator<<(std::ostream &out, Record &rec) {
     std::out<<'['<<rec.rid<<':'<<rec.content<<']';
}
```

Note: if you prefer, you can combine the two, and add the following to the class:

```
friend std::ostream& operator<<(std::ostream &out, Record &rec) {
     std::out<<'['<<rec.rid<<':'<<rec.content<<']';
}
```

However, even if you do that, as a friend function, it won't really be a method (that is, it isn't performed *by* the object, and don't rely on the `this` pointer).

If you want to overload the stream extraction (`>>`) as well, you certainly can, but automated reading from a stream is often less essential than a simple method of output.

Naturally, there are quite a few other operators we could also overload (some as friend functions; some as member functions), but very few would really make sense for this sort of record.

You're still encouraged to take a look at them sometime (the parenthesis operator is especially interesting).

For now, let's look at that record ID…

Step Six: Static Functions and Variables

The current version is fine, but there's a usage issue: ID values like that are typically *autoincrementing*. What we'd prefer would be to not need to specify the ID at all, and somehow automagically having sequentially-incrementing IDs assigned to each new Record.

Actually, we can do that (pretty easily)!

First, add this to the class:

```
static long ridCount; //Tied to the class; not the object
```

Because it's static, it's tied to the class; not any object. However, we can't initialize it here. So add this *after* the class:

```
long Record::ridCount=0;
```

Next, add the following to the class (probably under private):

```
static long nextRIDCount() {
     return ridCount++;
}
```

So, how does this help us? Remove the RID parameter from the constructor (leaving it with only content). Next, change `this->rid=rid;` to `this->rid=nextRIDCount();`

Step Seven: Assignment Operator

Try the following somewhere:

```
Record r("abba"),s("betta");
r=s;
```

It crashes. Why?
It's because that doesn't use the *copy constructor*; it uses *assignment operator*.
Why should that matter? Two problems:

1. Just as with the original copy constructor, we now end up trying to deallocate the same array twice
2. Before the current array is clobbered, we don't deallocate *its* memory at all

There are three solutions:

1. Properly implement an overloaded assignment operator:
   - Make sure to check if it's being assigned to itself! (Basically, check if *this* matches the address of the received parameter) If it is, simply return `*this`
   - Deallocate the old memory, then allocate new memory to hold the contents of the parameter
   - Copy over the contents
2. Use the older C++ approach of overloading the assignment operator, but don't give it any actual content, and simply put it under private
   - e.g. `Record& operator=(const Record &other) {}`
3. Use the C++11 (and above) approach of *deleting* the assignment operator
   - e.g. `Record& operator=(const Record &other) = delete;`

Pick any of the three. I'll be assuming #2 from this point onward.

Step Eight: Organization

You now have a functioning class. However, it's cluttering up the main source file. Ideally, we'd like to be able to separate it out.

Create two new files: Record.cpp and Record.h.
At the beginning of your main source file, add `#include "Record.h"`
Next, we wish to move all of our class into those two files.
For Record.h:

```
#include <iostream>

class Record {
private:
	long rid; //Record ID
	char *content;

	static long ridCount; //Tied to the class; not the object
	static long nextRIDCount();
	Record& operator=(const Record &other) {}

public:
	Record(char content[]);
	Record(const Record &original);
	~Record();
	long getRID();
	char *getContent();
	friend std::ostream& operator<<(std::ostream &out, Record &rec);
};
```

For Record.cpp:

```
#include "Record.h"

long Record::nextRIDCount() {
	return ridCount++;
}
long Record::ridCount=0;

Record::Record(char content[]) {
	int clength=0;
	this->rid=nextRIDCount();
	while (content[clength]!='\0') clength++;
	clength++;
	this->content=new char[clength];
	for (int i=0;i<clength;i++) this->content[i]=content[i];
}

Record::Record(const Record &original):rid(original.rid) {
	int clength=0;
	while (original.content[clength]!='\0') clength++;
	clength++;
	this->content=new char[clength];
	for (int i=0;i<clength;i++) this->content[i]=original.content[i];
}

Record::~Record() {
	delete[] content;
}

long Record::getRID() {
	return rid;
}
```

```
char *Record::getContent() {
      return content;
}

std::ostream& operator<<(std::ostream &out, Record &rec) {
      std::out<<'['<<rec.rid<<':'<<rec.content<<']';
}
```

As mentioned above, in your main source file, remember to `#include "Record.h"`.

Note that your g++ line has changed a little. Assuming a main source file of inagaddadavida.cpp:

```
g++ -o inagaddadavida inagaddadavida.cpp Record.cpp
```

(As mentioned in lecture, there's no need to include Record.h, as that's what the #include lines are for)

Submission Task

Write a very simple program to demonstrate that you can make use of a couple of Records. All you need do is output a couple different records, with auto-incrementing record IDs.

Submission

To demonstrate that you understand the content from this lab, simply show the output of your program for the submission task to your lab demonstrator.