Lecture Summary

In this lecture, we learned about various useful C++ constructs. We learned how to handle errors using exceptions, the role of different constructors (in particular copy constructors), and operator overloading.

1 Handling errors

In the previous lecture, we learned about Stacks and Queues. We formally defined the operations; in particular, for Stacks, we defined exactly what the semantics of \texttt{top()}, \texttt{pop()} and \texttt{push()} were. You may remember that those semantics did not include what should happen if \texttt{top()} was to be called on an empty stack. So suppose that we had implemented a class \texttt{IntStack} that captures a stack of integers. Now someone else who got hold of our code writes the following:

```cpp
IntStack *s = new IntStack;
cout << s->top() << endl;
```

When \texttt{top()} is called, the Stack will still be empty. The default implementation of our stack (using linked lists, say) implemented \texttt{top()} as follows:

```cpp
int IntStack::top()
{
    return head->data;
}
```

The result here will be an attempt at dereferencing a NULL pointer, triggering a segfault. The program will quit, and the person using our Stack will not really know what went wrong (except after some debugging). Also, simply ending the program may not be the right thing to do here. What’s the right way to deal with this problem? Several solutions suggest themselves:

- **Hold programmers responsible for safe use of the data structure.** This falls into the broader paradigm “garbage in, garbage out”. This is not a completely unreasonable position to hold; the person wanting to use your class should make sure to test whether the stack is empty before accessing it. But as we all know, “should” does not mean “will”, and your goal when writing code is not necessarily to help others develop good habits, but to develop good code yourself.

- **Have bad calls to \texttt{top()} return a ‘magic’ value like -1.** This might work if the stack is known to only contain positive values; the programmer could then use the unexpected return to debug their code. However, if the stack can contain any integers, there is no “safe” magic value. And for a more general stack (of a templated type), this solution is untenable, as we wouldn’t even know what to return. Further, the programmer might simply choose to ignore bad returns, which would cause errors to trickle up.

- **Use \texttt{assert} statements.** In general, \texttt{assert} statements are good debugging technique. For instance, we could insert the statement \texttt{assert (head!=NULL)} as a first line of \texttt{top()} and \texttt{pop()}. (To use \texttt{assert} statements, you need to \#include\texttt{<cassert>}.)


An assert statement simply causes the program to crash when the asserted condition fails to hold. While this is still a crash, at least, it gives more useful information, namely, printing in which line the crash happened, and what condition failed. This helps significantly in debugging.

However, the program still crashes, which means that it does not have the option of dealing with the error themselves.

These solutions all still involve crashes, which may not be the best way to handle mistakes, e.g., if you’re running the backend database for a major website, or the control software for a rocket or a nuclear power plant. Instead, it would be better to actually signal to the caller of our function that something went wrong (and what), and give it a chance to handle the mistake.

The way this is traditionally done in C is by making the function return a bool instead of the actual value we’re interested in, and passing the actual value by reference in another variable. The following example illustrates this:

```c
bool IntStack::top (int& v)
{
    if (head != NULL) {
        v = head->data;
        return true;
    } else return false;
}
```

This is a significant improvement. A slight downside is that the calling function may still ignore the bool value, and simply use the undefined value in v.

The C++ solution is to use exceptions. Exceptions allow us to provide feedback about the success of an operation, in such a way that a caller must deal with the failure condition explicitly. The basic use of an exception is as follows:

1. Upon failure, the method notifies the calling code by “throwing” an exception.
2. This exception propagates “up” through the program stack until it reaches a piece of code designed to handle it; it continues there.
3. If no such code is found, the program terminates.

Any function that could throw an exception somewhere inside but cannot process the exception within the function itself must declare itself as an exception-throwing function. This allows the exceptions to trickle upward to potentially be handled by a higher level of calling code.

As an illustration, a revised version of our top() function may look as follows:

```c
// header
// Functions that throw exceptions need to declare so in header.
int top() throw(logic_error);

// implementation
int IntStack::top() throw (logic_error) {
    if (head == NULL) throw logic_error ("head pointer was null!");
    else return head->data;
}
```

Whenever you want to write code that throws/handles exception, you should #include<exception>. In principle, any kind of object (even an integer, or an IntListElement) can be “thrown” and treated as an exception. However, typically, you would like to only use objects that inherit from the class exception, since they contain some standard methods that tell you about what caused the exception. C++ also provides
a few standard “types” of exceptions pre-implemented in #include<stdexcept>; for instance, the class logic_error we used above is included in that header file. Throwing a type of exception whose name (and content) reflect the type of error that happened helps in understanding what error occurred.

At the point at which an exception can be reasonably handled, we use the try-catch construct to deal with it. try is used for a block that may throw an exception; catch provides code to execute in case an exception actually does happen. If a piece of code would not know what to do with an exception, it should not catch it, and instead let it percolate up to a higher level, where it may be reasonably handled. In our earlier stack example, the main function may now look as follows:

```cpp
//main
try {
    cout << s->top() << endl;
    cout << "Printed successfully!" << endl;
} catch (logic_error &e) {
    cout << "A logic error occurred!" << endl;
    cout << e.what();
}
```

Here, if/when an exception occurs, the execution of the try block terminates (so we never get to the second line about printing successfully); instead, the program jumps to the first catch block that matches the exception. In our case, because a logic_error is thrown, the catch block matches, so the program prints the message. It then also prints the string we had passed into the constructor (i.e., “head pointer was null”), since that’s what it returned by the what() function in the class exception. The existence of the what() function is one of the reasons to use classes derived from exception. If you write your own exception class, you should inherit from exception and overload the virtual what() function to report what your exception is about.

Notice in the above example that e is an exception object, which is passed by reference. It has some data in it. For example, we just put a message string in it, but we could also have put in the description of the variables that caused the exception. We could write our own exception object to throw as well; for instance, in implementing a List class, we may want to have an exception if the index to access is out of bounds.

```cpp
class OutOfBoundsException : public exception {
  public:
    int pos;
    EmptyStackException (int d) {
      pos = d;
    }
  }

  int List::get (int pos) throw (OutOfBoundsException) {
    if (!inRange(pos)) throw OutOfBoundsException (pos);
    else return a[pos];
  }
```

A try block can have multiple catch blocks associated with it, to deal with different types of exceptions. In this case, the program will execute the first catch block that matches the exception that is thrown. This means that the more specific types of exceptions should precede the more general ones.

```cpp
List *L = new List ();
try {
    cout << L->get(3);
}
```
catch (OutOfBoundsException &e) {
    cout << "Array Index was out of Bounds" << endl;
    // specific treatment when index is out of bounds
}
catch (exception &e) {
    cout << "General Type of exception" << endl; }

Here, the first block may have some treatment for specifically the case when the array index was out of bounds, while the second block may have a less specific treatment (such as just an error message) when some other, less specific, error occurred. If we had put the two catch blocks in the opposite order, the special treatment for OutOfBoundsException would never kick in, as all those exceptions would already be caught in the more general block.

2 (Copy) Constructors

When we started talking about classes and objects, we also mentioned constructors briefly. It’s worth mentioning explicitly now that a single class can have many different constructors, so long as their signatures (number or types of arguments) are different. Two types of constructors that can be quite frequently useful are the following:

class A {
    A (string s);
    A (const A & otherA);
}

The former would be useful for parsing a string s and putting the data into an object of type A. The second constructor is called a copy constructor. It is useful for creating an identical copy of an element. With this constructor, we can “copy” objects as follows:

A *a1, *a2;
a1 = new A("Hello!");
a2 = new A(a1);

3 Operator Overloading

In an effort to make our code easy to read, we often would like to use (unary and binary) operators. For instance, if we define our own way of testing equality between two objects of a certain class, it would be nice to be able to write if (obj1 == obj2) instead of if (obj1.equals(obj2)), if we were to define a function equals(). In a sense, this is just “syntactic sugar” — it makes the code prettier, or easier for humans to read, but does not really give us functionality we didn’t have before.

C++ allows us to overload the meaning of operators, such as ==, !=, = (assignment), +, -, *, [], and many others. The ones perhaps most frequently overloaded are assignment and equality test, i.e., = and ==.

The syntax for operator overloading is as follows:

class A {
    // for binary operators
    // pattern: [RETURNTYPE] operator[OPERATOR] ([RHS DATA])
    // Example:
    bool operator==(const A & otherA) {
        // return this->data == otherA.data
    }
}
// for unary operators
// pattern: [RETURNTYPE] operator[OPERATOR]
// Example:
A& operator++ () {
    // increment A’s data field.
    return *this;
}

Notice that [RHS DATA] captures the second operand of the operator. The operator syntax allows us to use the following shorthand:

A a1, a2;
// the following two lines of code are exactly the same thing
if ( a1 == a2 ) cout << "They’re equal!";
if ( a1.operator== (a2)) cout << "They’re equal!";

So by using the weird name operator== of the function, we’ve made it possible to use the nice-to-read shorthand.

The other type of operator we frequently overload is the assignment operator, as in the following abbreviated example:

A& operator= (const A& otherA) {
    // A.data = otherA.data
    // copy everything from otherA to A.
    return *this;
}

The reason an assignment operator should be of type A& rather than void is to allow us to write a statement such as a = b = c, which we routinely do with other types. For that reason, when we write b=c, what comes out must also be of the type that can be assigned to a.

In writing a copy constructor, we have to be a little careful. If our class A has dynamically allocated data (such as an internal array), then to avoid memory leaks, those data need to be deallocated before we copy over the stuff from the other object. But that could get us into trouble when writing a statement a = a. While this is not necessarily a very useful statement to write, you wouldn’t want it to break the other person’s code because of your wrong implementation of the operator. So what you should do is first check that you’re not assigning an object to itself; once you’ve ensured that, it’s safe to deallocate the old data in the object you’re assigning to, then copy over the data from the other object.

3.1 An Illustration: Complex Numbers

The following illustration was not covered in class — instead, it was covered in lab. But it shows in more detail how to actually code with operator overloading (whereas the preceding discussion was a bit abstract).

Suppose that we want to implement a class for complex numbers. Recall that complex numbers are of the form \(a + ib\), where \(a, b\) are real numbers, and \(i\) is such that \(i^2 = -1\). The addition of complex numbers is defined as expected: \((a + ib) + (c + id) = (a + c) + i(b + d)\).

We can use the following definition of a very basic Complex class:

class Complex {
private:
    double re, im;
public:
    Complex (double re, double im) {

this->re = re;
this->im = im;
}

Complex (const Complex & toCopy) {
    // let's add a copy constructor while we're at it.
this->re = toCopy->re;
this->im = toCopy->im;
}

Complex add (Complex other) {
    double reSum = re+other.re;
    double imSum = im+other.im;
    return Complex(reSum, imSum);
}

string toString() {
    return (""+re+ "+" +im+"i");
}

Now, we could add two Complex numbers and print the sum using the following code:

Complex c1 = Complex(1.5,-3.2);
Complex c2 = Complex(1.1,1.3);
Complex sum = c1.add(c2);
cout << sum.toString() << endl;

While this works well, it is a little tedious to have to write function calls like add() and toString(). This makes comprehending the code more difficult than it needs to be. To get around this, we can use the idea of operator overloading we just saw. Instead of (or in addition to) having a function add(), we could overload the operator +, as follows:

Complex operator+ (const Complex& other) {
    return add(*this, other);
}

Now, we can also perform the third line of the code above using the shortcut:

Complex sum = c1+c2;

This is exactly equivalent to the previous segment, but is much easier to interpret. We can now overload a few more commonly used operators, to get a hang of it.

Complex operator- (const Complex & other) {
    double reDiff = re - other.re;
    double imDiff = im - other.im;
    return Complex(reDiff, imDiff);
}

Complex operator* (const Complex & other) {
    // look up multiplication of complex numbers if you don't remember it.
    double reProd = re*other.re - im*other.im;
    double imProd = re*other.im + im*other.re;
    return Complex(reProd, imProd);
}
return Complex(reProd, imProd);
}

boolean operator==(const Complex & other) {
    return (re==other.re && im== other.im);
}

boolean operator!=(const Complex & other) {
    return !(this == other);
}

boolean operator< (const Complex & other) {
    double absSq = re*re+im*im;
    double otherAbsSq = other.re*other.re + other.im*other.im;
    return (absSq < otherAbsSq);
}

boolean operator<= (const Complex & other) {
    return (*this<other) || (*this==other);
}

boolean operator> (const Complex & other) {
    return !(this<=other);
}

boolean operator>=(const Complex & other) {
    return !(this<other);
}

With these definitions in place, we could now do arithmetic and comparisons on our nice Complex class. What we can’t do yet is write things like cout << sum or cin >> c, which would be nice. In fact, C++ does allows us to overload the I/O operators « and ».

However, the implementation has to be a little different. If you look at the binary comparison operators above, you might notice that they were implemented as non-static member functions of the left-hand side of the operation, with the right-hand side being a parameter to the operation. However, in the operation «, this cannot be done, since the left hand side (cout) is not a member of the Complex class:

Instead, we have to declare the operator overload as a free function that takes an ostream and a Complex as parameters:

class Complex {
    //Implementation...
};

ostream &operator<<(ostream &out, const Complex & c) {
    out << c.toString();
}

The reason why the type of the operation is ostream (rather than void) is the same as for the assignment operator: we can now write code including multiple « in one statement, as follows:

Complex c1 = Complex(1,2);
Complex c2 = Complex(5,7);
cout << c1 << " + " << c2 << " = " << c1+c2 << endl;