CS 246 F13 Midterm Review

October 22, 2013

Agenda

- Shell
- C++
- Questions

Regular Expressions

- ˆ - match the beginning of line
- $ - match the end of line
- . - match any character
- ∗ - match the preceding pattern 0 or more times
- + - match the preceding pattern 1 or more times
- [. . .] - match any character in the set
- [ˆ . . .] - match any character not in the set
- $a|b$ - match either expression a or b
- Parentheses are used to override precedence

Regex Examples

Provide a regex:

- For lines starting with abc and ending with xyz

Regex Examples

Provide a regex:

- For lines starting with abc and ending with xyz
  - `^abc.*xyz$`

Regex Examples

Provide a regex:

- For lines starting with abc and ending with xyz
  - `^abc.*xyz$`
- For lines that assign either 0 or 1 to the variable count.

Regex Examples

Provide a regex:

- For lines starting with abc and ending with xyz
  - `^abc.*xyz$`
- For lines that assign either 0 or 1 to the variable count.
  - `count *= *(0|1)`

Regex Examples

Provide a regex:

- For lines starting with abc and ending with xyz
  - `^abc.*xyz$`
- For lines that assign either 0 or 1 to the variable count.
  - `count *= *(0|1)`
- For lines consisting of characters not in the set: {a,b,c,t,j,4,8,w}.

Regex Examples

Provide a regex:

- For lines starting with abc and ending with xyz
  - `^abc.*xyz$`
- For lines that assign either 0 or 1 to the variable count.
  - `count *= *(0|1)`
- For lines consisting of characters not in the set: {a,b,c,t,j,4,8,w}.
  - `^[^abctj48w]+$`

Shell Commands

Be familiar with basic commands.

For example, cat, wc, man, chmod, ls

It also pays to be familiar with common options

For example, ls -al or wc -l

I/O Redirection

Recall that 1 refers to stdout and 2 refers to standard error.

What do the following do?

- `./script 1> file`
- `./script 1>&2`
- `./script 2> file`
- `./script < file`

I/O Redirection

Consider the difference between:

```
$ wc -l myfile
4 myfile
```

and

```
$ wc -l < myfile
4
```

Tests

Used in if statements, while loops, etc.

Recall that the following have the precedence from highest to lowest:

- `!` has highest precedence
- `\( expr \)`
- `\( expr1 -a expr2 \)`
- `\( expr1 -o expr2 \)` has lowest

Tests

- For strings: =, !=
- For integers: -eq, -ne, -ge, -gt, -le, -lt
- For files: -d, -f, -e, -r, -w, -x

Tests

- For strings: =, !=
- For integers: -eq, -ne, -ge, -gt, -le, -lt
- For files: -d, -f, -e, -r, -w, -x

Some examples:

- `[ \( -f file -a -x file \) -o -w file ]`
- `[ "cat" != "dog" ]`
- `[ 5 -lt 6 -a 6 -ge 6 ]`

Quoting

- Backslash(\): escapes any character (gives literal meaning)
- Backquote(`): executes text as command

  ```
  $ echo ‘wc -l text‘
  4 text
  ```

- Single Quote(´): treat contents literally
  - No shell variable substitution

  ```
  $ foo=bar
  $ echo ’$foo’
  $foo
  ```

- Double Quote(”): recognizes escapes, backquotes, and variables

  ```
  $ foo=bar
  $ echo "$foo"
  bar
  ```

Script Example

```
#!/bin/bash
usage(){
echo "$0 file-name" 1>&2
}
count1=0 #number of words that are ’Hello’
count2=0 #number of words that are not ’Hello’
if [ $# -ne 1 ]; then
usage; exit 1
fi
for word in ‘cat "$1"‘; do
if [ "$word" = "Hello" ]; then
count1=$((${count1}+1))
else
count2=$((${count2}+1))
fi
done
echo "Hello appeared $count1 times."
echo "Non-Hello words appeared $count2 times."
```

C++

What we will not cover:

- Control structures from C (if, while, for)
- Structures
- C I/O and Memory Management

What we will cover:

- C++ Strings
- I/O Streams
- Pointers and References
- Dynamic Memory Management
- Overloading
- Basic Classes

Strings

- To use:

  ```
  #include <string>
  ```

- Encapsulates the idea of a C-string in a class
- Access individual characters like a C-string

  ```
  string str = "abc123";
  str[3] = 'z';
  ```

- Has useful methods:
  - substr
  - length
- Has overloaded versions of:
  - operator+
  - comparison operators (e.g. $<$)

I/O Streams

```
int x,y;
cin >> x;
cin >> y;
cout << x << " and " << y << endl;
```

What is printed given the following input?

- 123,456
- 123 456

Recall that << and >> are overloaded for all POD types (int, double, char) and string.

I/O Streams: End of File

- Stream member `eof()` returns true if end of file is reached
- Stream member `fail()` returns true if end of file or an invalid token is seen
- Streams can be used as part of conditional tests
    - Due to an implicit conversion to void*

```
while(cin >> x)
{
  sum += x *2;
}
```

I/O Streams: Files

- To use:

  `#include <fstream>`
- `ifstream` reads from a file
- `ofstream` writes to a file
- Works almost exactly like `cin` and `cout`

```
ifstream ifs ("file.in");
ofstream ofs ("file.out");
// Check if files were opened:
if (ofs.fail() || ifs.fail()) cerr << "Files not opened\n";
int x;
ifs >> x;
ofs << x;
```

Pointers and References

Consider:

```
int x = 42;
int &y = x;
int *z = &y;

//What is printed?
cout << boolalpha;
cout << x==z << " ";
cout << &x==y << " ";
cout << &x==z << " ";
cout << x==y << " ";
cout << &x==&z << "\n";
```

Pointers and References

Consider:

```
int v = 42;
const int w = v;
const int *x = &v;
int * const y = &v;
const int * const z = &v;
// Which of the following lines cause an error?
int *a = &v; // 1
int *b = &w; // 2
*x = 50;     // 3
*y = 50;     // 4
y = &w;      // 5
```

Pointers and References

Pass by value:

```
void foo(int x);
void foo(int *x);
```

Pass by reference:

```
void foo(int &x);
void foo(const int &x);
```

Question: What is the benefit to pass by const-ref?

Overloading

- Occurs when a name has multiple meanings in the same context
- *Most* languages allow some level of implicit overloading
  - e.g. operator+ for integers, floats, and strings
- In C++:
  - Number and type of parameters are used to select which function to use
  - Return type is never considered

Overloading

The following qualifiers do not make a parameter unique:

- signed
- const
- &

Which of the following are *valid* overloads?

```
void r(int i);
void r(signed int i);  // 1
void r(const int i);   // 2
void r(int& i);         // 3
int r(int i);           // 4
void r(unsigned int i); // 5
void r(int i, int j);   // 6
void r(long int i);     // 7
```

Operator Overloading

- Operators are just like functions except that they are used infix

  `a+b is actually operator+(a,b)`

- This implies we can overload them like we would functions

  ```
  MyStruct operator+ (MyStruct &a, MyStruct &b)
  {
    ...
    return newStruct;
  }
  ```

Operator Overloading: Streams

- Two I/O operators to overload:

  ```
  istream& operator>> (istream& is, <type> var);
  ostream& operator<< (ostream& os, <type> var);
  ```

- Note that istream works for all kinds of input streams
- Why should we always remember to return the same stream that is passed in?

Operator Overloading: Streams

- Two I/O operators to overload:

  ```
  istream& operator>> (istream& is, <type> var);
  ostream& operator<< (ostream& os, <type> var);
  ```

- Note that istream works for all kinds of input streams
- Why should we always remember to return the same stream that is passed in?
  - Allows cascading

Operator Overloading: Cascading

```
myClass a,b,c;
cin >> a >> b >> c;
//Equivalent to:
((cin >> a) >> b) >> c;
```

- Call overloaded `operator>>` for `myClass`
- Want to return continuously from cin (not some other stream)
- Some other operators also cascade (assignment, arithmetic, etc)

Dynamic Memory Management

- C++ provides dynamic memory allocation with `new`, `new []`, `delete`, and `delete []`
- C uses `malloc` and `free`, which you should not mix with C++ memory management
- Memory for dynamic allocation comes from the heap
- `new` will generate an error (and your program will crash) if the heap is full
- Deallocate memory when it is no longer need

```
int * parr = new int[10];
...
delete [] parr;
```

Dynamic Memory Management

- In C++, you have the choice of allocating on the stack or on the heap
- Stack allocations eliminates explicit memory management and is more efficient than heap allocation
- Dynamic allocation should use be used when memory must outlive the scope in which it was created
- **Remember:** Returning pointers/references to stack based memory is bad

From Structs to Classes

- Recall that:
  - **Structure**: groups related data together
  - **Class**: groups related data and methods that operate on that data together
  - **Object**: is an instance of a class
- Using classes allows us to abstract away from an implementation and rely on an interface that can be separate from that implementation

Classes: Methods

- A class has member routines (methods) and member variables (fields)
- Methods can access fields without qualification (almost always)
- However, every method has an implicit `this` pointer that points to the invoking object
- The `this` pointer can be used to disambiguate fields from parameters with the same name

  ```
  struct Rational{
    int numer, denom;
    void setNumer(int numer){
      this->numer=numer;
    }
  };
  ```

Classes: Constructors

- A constructor is a special member routine used to implicitly perform initialization after an object is allocated
- A `default` constructor is one which takes no parameters
- A constructor takes the class name and can be overloaded in the usual fashion
- `const` fields and `references` must be initialized before they can be used
- What mechnaism do we use to solve this problem?

Classes: Constructors

- A constructor is a special member routine used to implicitly perform initialization after an object is allocated
- A `default` constructor is one which takes no parameters
- A constructor takes the class name and can be overloaded in the usual fashion
- `const` fields and `references` must be initialized before they can be used
- What mechnaism do we use to solve this problem?
  - Initialization list

```
struct Student {
  const int id;
  string name;
  Student(int id, string name)
    : id(id), name(name){}
};
```

Classes: Destructors

- Has no return type, prefix class name with ˜
- Used to “destroy” an object
- Typically, this involves deallocating memory and any other clean up

```
struct VarArray{
  int * arr;
  int size;
  VarArray(int n) : size(n), arr(new int[n]){}
  ˜VarArray()
  {
    if(arr != NULL) delete [] arr;
  }
};
```

Classes: Copying Objects

There are multiple contexts where an object is copied:

1. Declaration initialization
2. Pass by value
3. Return by value
4. Assignment

```
MyObject o1,o2; // Default ctor
MyObject o3 = o1; // Case 1
o2 = o1; // Case 4

void foo(MyObject o); // Case 2

MyObject foo(); // Case 3
```

The first 3 cases involve a newly created object.

The fourth cases involves an existing object.

Classes: Copy Constructor

- The first 3 cases all invoke the copy constructor for a class.
- The copy can be either **deep** or **shallow**
  - **Shallow** will copy pointers
  - **Deep** will allocate new memory and copy values

```
struct Shallow{
  int *i;
  Shallow (int v) { i = new int; *i = v;}
  Shallow (const Shallow& o)
    : i (o.i) {}
};
struct Deep{
  int *i;
  Deep (int v) { i = new int; *i = v;}
  Deep (const Deep& o)
    : i (new int)
  {
    *i = *(o.i);
  }};
```

Classes: `Assignment Operator`

- Is invoked in the fourth case
- Copies values of an object into an existing object (may need to allocate more memory)
- Be careful of self-assignment
- Copy-and-swap idiom prevents many errors
  - Requires copy constructor and destructor

Classes: Assignment Operator

```
struct VarArray{
  int * arr;
  int size;
  VarArray(int n) : size(n), arr(new int[n]){}
  ...
  VarArray& operator=(const VarArray& o)
  {
    if(this == &o) return *this;
    int * tarr = new int[o.size];
    delete [] arr;
    arr = tarr;
    size = o.size;
    for(int i=0; i<size; ++i)
    {
      arr[i] = o.arr[i];
    }
    return *this;
  }};
```

Classes: Assignment Operator

Using copy-and-swap:

```
struct VarArray{
  ...
  void swap(VarArray &o){
    int *tarr = o.arr;
    o.arr=arr;
    arr = tar;
    int tmp = size;
    size = o.size;
    o.size = tmp;
  }
  VarArray& operator=(const VarArray& o)
  {
    VarArray tmp = o;
    swap(tmp);
    return *this;
  }
};
```

Rule of Three

The **Rule of Three** is a rule of thumb that states:

```
If a class defines a destructor, copy constructor, or
assignment operator, then it probably requires all
three.
```

Member Operators

- Operators can be methods of a class
- LHS of operator is `this`
- Why should >> and << not be member operators?

FIN

Questions