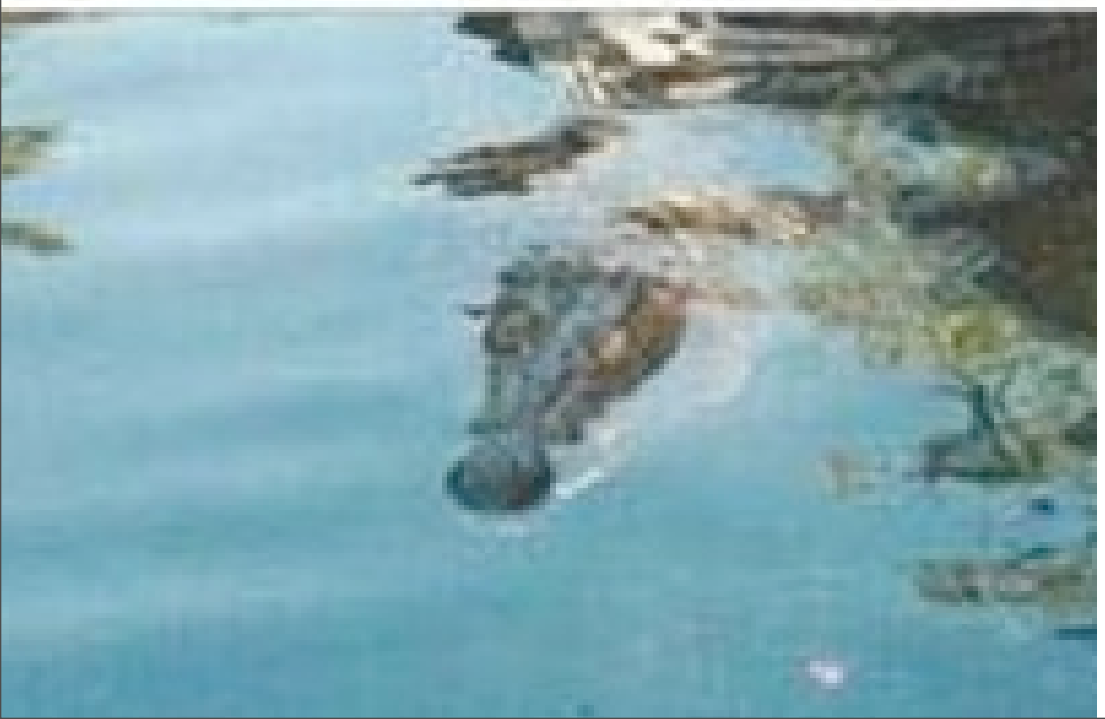
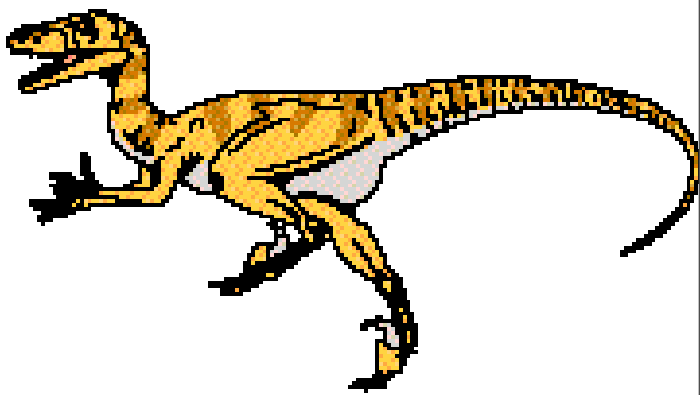

THOSE WHO
THROW OBJECTS
AT THE CROCODILES,
WILL BE ASKED TO
RETRIEVE THEM

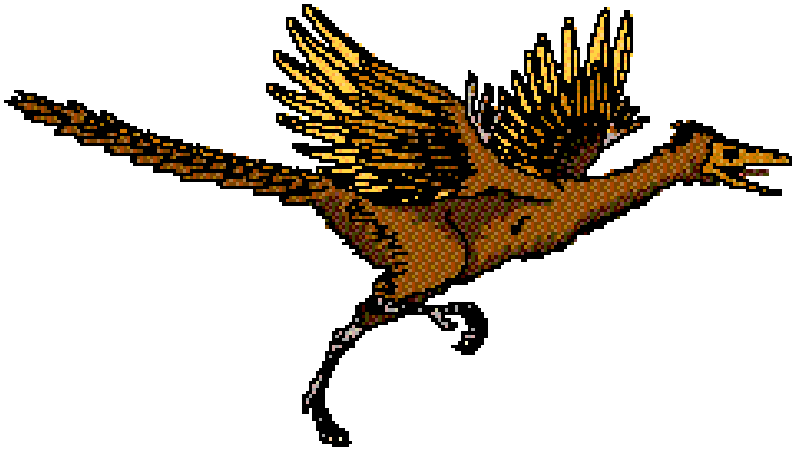


IOSTREAMS

Review

- What does **dynamic_cast** let you do? Why is it sometimes preferable to C-style casting?
- How do you throw an exception?
- How do you catch an exception?
- Once an exception is caught, where does execution pick back up?

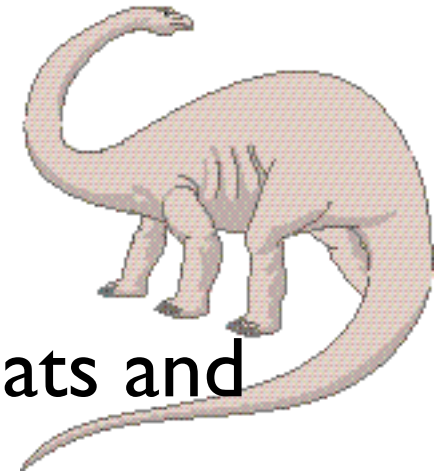




The Basics

- I/O is a big part of nearly every program
- We've been doing simple I/O for most of the semester, using **cin** and **cout**
- **cin** and **cout** are just two examples of a more general C++ feature called **iostreams**

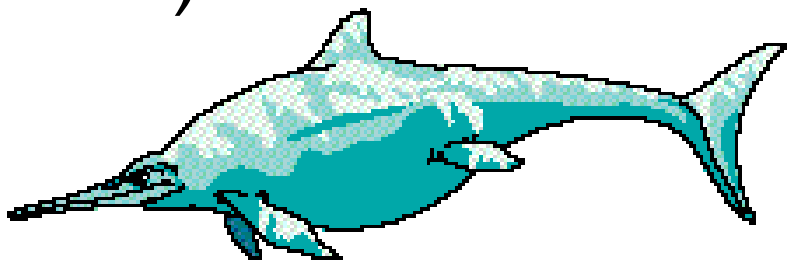
Streams



- A **stream** is a C++ object that formats and holds bytes of data
- There can be input streams (an **istream**) or an output stream (an **ostream**)
  - cin is an istream, cout is an ostream; these give you access to **stdin** and **stdout**
- Streams don't only do I/O: they also buffer the data to make I/O more efficient

iostream properties

- Streams are designed to be source independent: a stream should be used the same way regardless of where the data is coming or going
- The same interface can work on:
  - keyboard/screen I/O (cout/cin)
  - file I/O
  - network I/O
  - a string
- Thanks to the magic of... ?

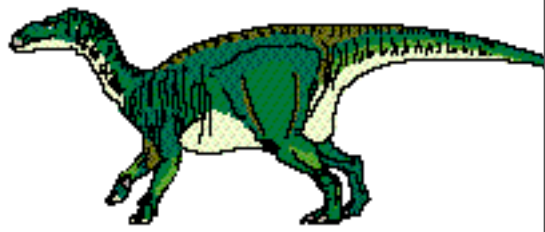


- We've been doing stuff like this all semester:

```
int input;

cin >> input;
cout << "this is some output" << endl;
```

- Let's look at what this stuff actually *is*:
  - **>>** is an *extraction* operator
  - **<<** is an *insertion* operator
  - **endl** is a *manipulator*
  - **cout** is an *ostream*; **cin** is an *istream*

Manipulators



- A *manipulator* is an object that acts on the stream itself
- **endl** is an example: when we try and "print" an endl:
  - it inserts a newline into the stream
  - it flushes the stream
- There's a bunch of other manipulators that we can use too

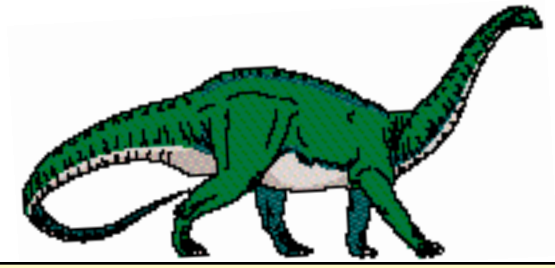
More Manipulators

- We can just flush the stream, *without* printing a newline first:

```
cout << flush;
```

- We can change the number base to oct (octal) or dec (decimal) or hex (hexadecimal) to any subsequent integers will be output in that base:

```
cout << hex << "0x" << i << endl;
```



Input

```
int i;
cin >> i;

float f;
cin >> f;

char c;
cin >> c;

char buf[100];
cin >> buf;
```

- Input tends to be fragile
- Users have to input the right data types, in the right order
- If the input isn't what the program expects, it can choke
- This is true with iostreams too:

What does this code do with this input?

```
12 1.4 c this is a test
```

The Problem

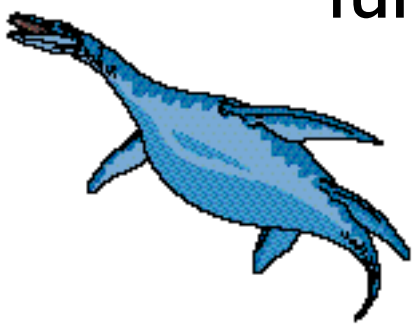
- By default, istreams are *space delimited* (as you may have seen in some of the projects)
- So when we attempt to do something like this:

```
char buf[100];
cin >> buf;
```

- with the input "this is a test", buf will contain the word "this"
- The rest of the input stays buffered

reading in a whole line

- Often you'll need to read in entire lines (until there's a newline character in the input stream)
- You do this using the **getline** member function:



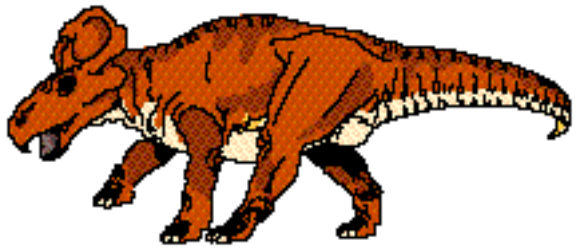
```
char buf[100];
cin.getline(buf, 100);
```

Note that we have to give cin a size, too! (why?)

Getting a character

- Another way to do things:
- Sometimes you want to get input character by character (*including* the whitespace!)
- You can do that with another cin member function:

```
cin.get();
```

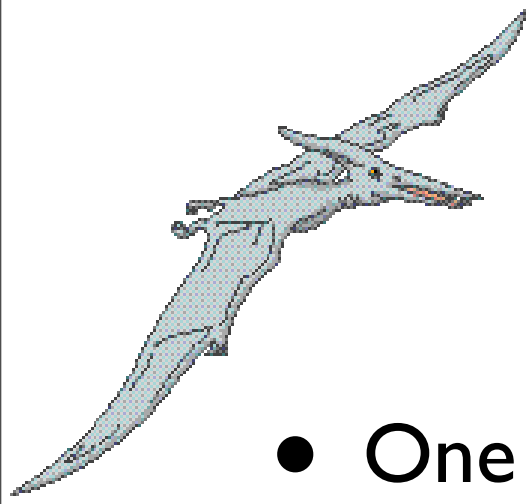


- get() reads the *next single character* from the stream (or EOF if the stream is at its end)

Streams Weirdness

- Input streaming doesn't always work the way you think it does
- How does this chunk of code act?

```
char answer;
cout << "Exit Program? [Y/N] ";
cin  >> answer;
cout << "Press Enter\n";
cin.get();
```

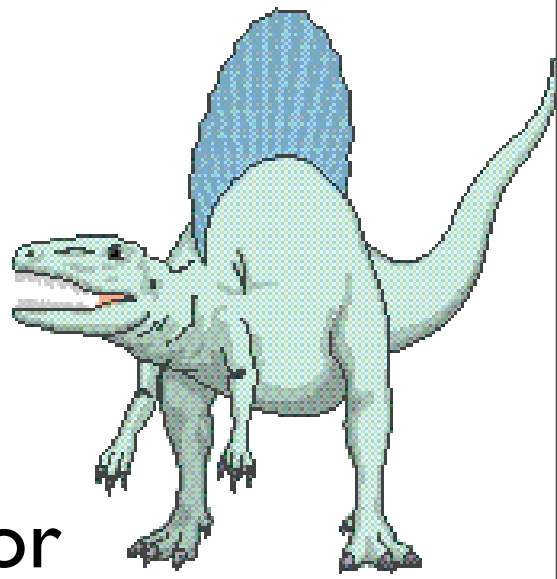


Discarding Input

- One solution: get rid of stuff in the stream buffer that we aren't going to want to deal with
- We can do this with the ignore() function:

```
cin.ignore();     // ignores a single character
cin.ignore(3);    // ignores 3 characters

// ignores 10 characters, or the "stop character",
// whichever comes first
cin.ignore(10, '\n');
```

File I/O



- So far we've used iostreams solely for console input/output
- A more important use is for file I/O
- This works largely the same way, although there's a bit more work required
- For file I/O, we must **#include<fstream>**

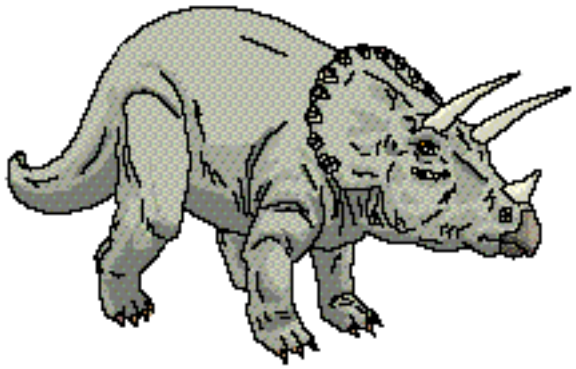
Starting Out

- To begin with, we create an object of the appropriate type: **ifstream** defaults to input, **ofstream** defaults to output
- We create the object and call good() on it to make sure it got instantiated properly:

```
ofstream output("c:\\test.txt");
if( !output.good() )
    return;
```

- At this point the object can be used much like cout or cin

Open Modes



- We can control the way a file is opened by changing an argument to the ifstream/ofstream constructor:

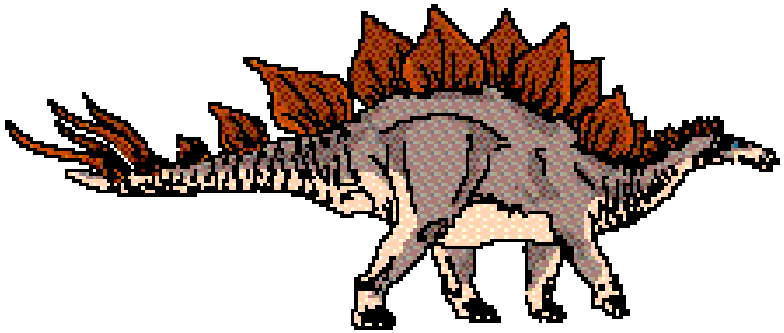
ios::in	open a file for input
ios::out	open a file for output (truncation)
ios::app	open a file for appending
ios::ate	open an existing file and seek to the end
ios::nocreate	open a file only if it *does* exist
ios::noreplace	open a file only if it does *not* exist
ios::trunc	open a file and delete the old one if it exists
ios::binary	open a file in binary mode (default is text)

Multiple Modes

- We can combine these flags by OR-ing them together with the **bitwise OR operator**: |

```
ofstream outFile("out.txt", ios::app | ios::nocreate );
```

- This opens "out.txt" for appending, and fails if the file doesn't already exist
- The | operator combines the different flags together - this is pretty common



...Seeking

- Each ofstream or ifstream has a read position and a write position - we ***seek*** through the file by changing these, so the object reads from/writes to a different spot
- We do this with the **seekg** (changes the get pointer) and **seekp** (changes the put pointer) member functions
- They let us seek relative to a position: the beginning, current position, or the end

Seeking Example

- We tell the seek function to seek x number of bytes relative to the beginning (ios::beg), current position(ios::cur), or end (ios::end) of the file

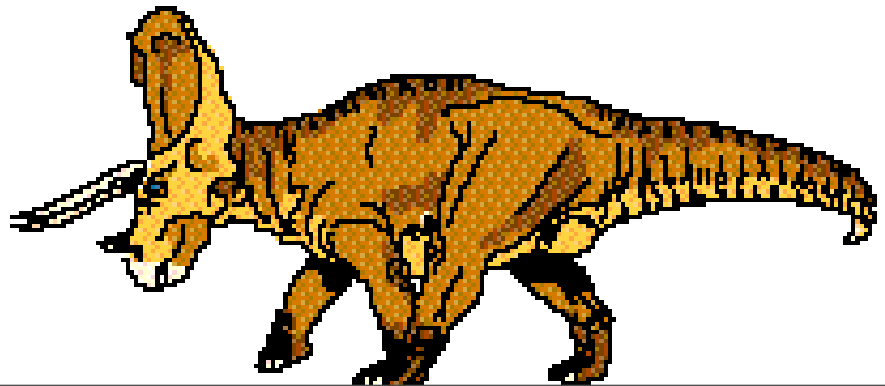
```
ifstream in("test.txt");
char c;

if( !in.good() )
    return;

// seek 50 bytes from the beginning of the file
in.seekg( 50, ios::beg );
in >> c;
```

Error Handling

- We can find out whether an iostream object is OK using a few member functions:
  - **eof()** returns true if the end of the file (or input) has been reached
  - **fail()** returns true if some operation has failed - formatting issues, for example
  - **bad()** turns true if something serious went wrong - running out of memory, for example
  - **good()** returns true if none of that stuff happened and everything is groovy

Error Handling 2

- To "reset" the error status of an iostream object, you can use the **clear()** function
- We might do this if we want to keep using the object - aka "rewind" a file and read some more from it
- clear() only resets the error status - it doesn't do anything with the buffer



Insertion/Extraction

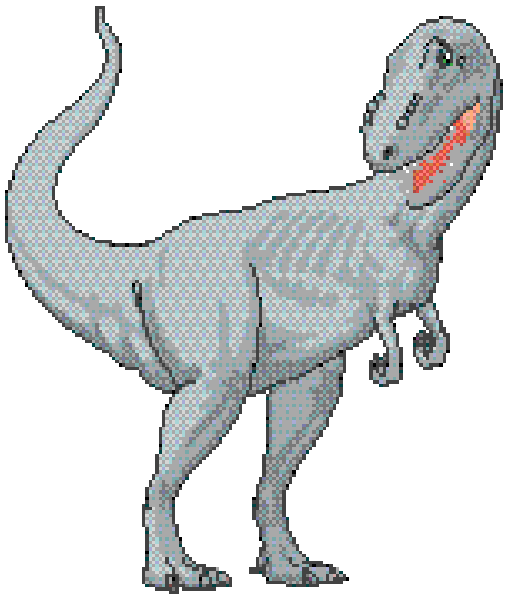
- iostreams are a library, not built into the language
- So << and >> don't have any special I/O meaning to the compiler - these are all overloaded!
- So for every data type that can appear on the right side of a >>, there's an overloaded **operator>>** function somewhere

```
int input;
cin >> input;
```

This works because istream defines an operator>> that accepts an integer as a parameter

Insertion/Extraction

- So far we haven't learned any way to make the following code work
- The **<<** operator is not defined for MyClass and ostreams, so this is a compiler error



```
class MyClass
{
    // stuff is declared here
};

MyClass m;
cout << m << endl;
```

- We can make it work providing that definition

Operator Overloading

- When we're overloading << and >> for our classes, these overloaded operators *can't be defined* as member functions!
- They still need access to private class data, though, so they're usually defined as global functions, and declared as friends
- Once we've overloaded << and >> for a custom class, we can use that class with iostreams such as cin/cout

```
class TwoInts
{
  public:
    TwoInts()
    { one = two = 17; }

    friend ostream& operator<<( ostream&, TwoInts& );
    friend istream& operator>>( istream&, TwoInts& );

  private:
    int one, two;
};

ostream& operator<<( ostream& out, TwoInts& ti )
{
   out << ti.one << ti.two;
   return out;
}

ostream& operator>>( istream& in, TwoInts& ti )
{
   in >> ti.one;
   in >> ti.two;
   return in;
}
```