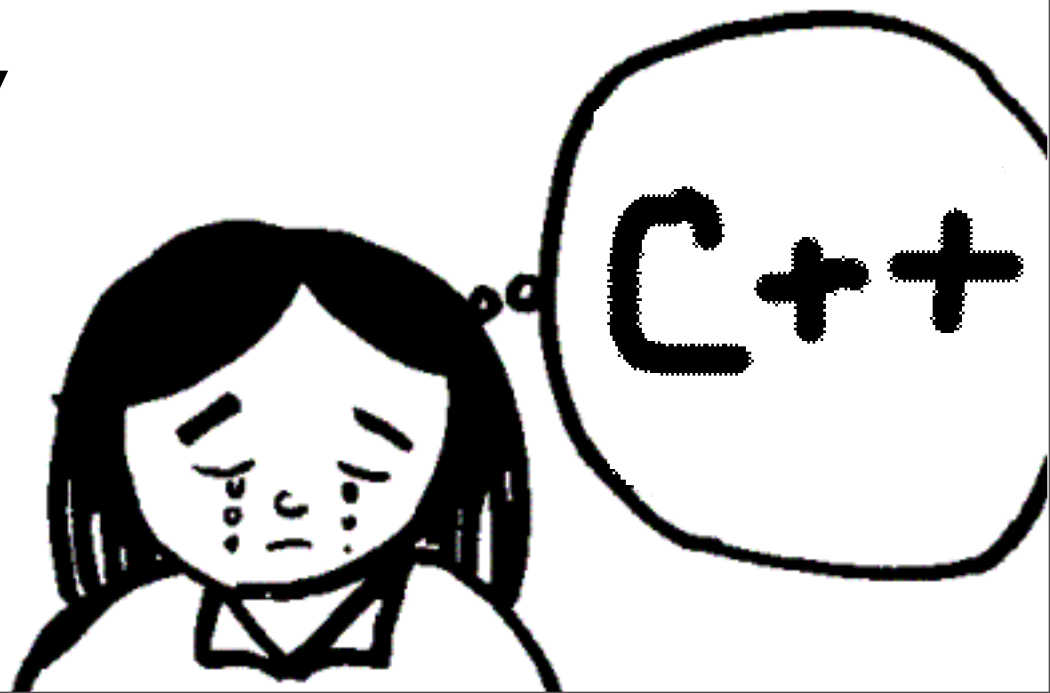
SMART POINTERS

Preprocessor Review

- What is an "include guard" on a header file? What does it look like?
- Write a SwapTimesTwo macro that accepts two arguments, swaps 'em, and multiplies them by two.
- Why are macros not good in C++? What should they be replaced with?

C++ Review

- What are the functions automatically included in every class if you don't specify them yourself?
- When working with inheritance, in what order are constructors/destructors called?
- What is "minimal evaluation"?
- How do you dynamically allocate an array of **Cat**'s, C-style?
- What does **virtual** mean in C++-land?

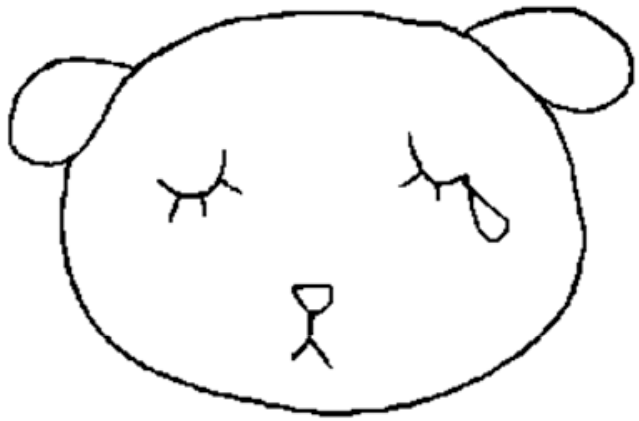


About Pointers

- Pointers are useful and necessary in C/C++
- ... and yet, a major cause of bugs!
  - dangling pointers, NULL pointers, memory leaks, and so on
- What can we do to keep the useful stuff but get rid of some of the error-prone stuff?

Smart Pointers

- We can make simple class objects act like pointers
  - Where have we seen something like this before? How did that work?
- What would you want a "smart pointer" to act like? How would you expect it to be implemented?



- A simple smart pointer class comes w/ C++: **auto_ptr**
- Part of the declaration is below...
- How does this class work?

```
template <class T> class auto_ptr
{
    T* ptr;
public:
    auto_ptr(T* p = 0) : ptr(p) {}
    ~auto_ptr()                 {delete ptr;}
    T& operator*()              {return *ptr;}
    T* operator->()             {return ptr;}
    // ...
};
```

```
void foo()
{
    MyClass* p(new MyClass);
    p->DoSomething();
    delete p;
}
```

(without **auto_ptr**)

- Here's the same code with and without using auto_ptr:
- What happens to **p** at the end of foo()?

(with **auto_ptr**)

```
void foo()
{
    auto_ptr<MyClass> p(new MyClass);
    p->DoSomething();
}
```

Why do we care?

- The big answer: *fewer bugs!* (hopefully)
  - automatic cleanup (fewer memory leaks)
  - automatic initialization (no garbage pointers!)
  - dangling pointers... what are these?

```
MyClass* p(new MyClass);
MyClass* q = p;
delete p;
p->DoSomething();    // Watch out! p is now dangling!
p = NULL;            // p is no longer dangling
q->DoSomething();    // Ouch! q is still dangling!
```

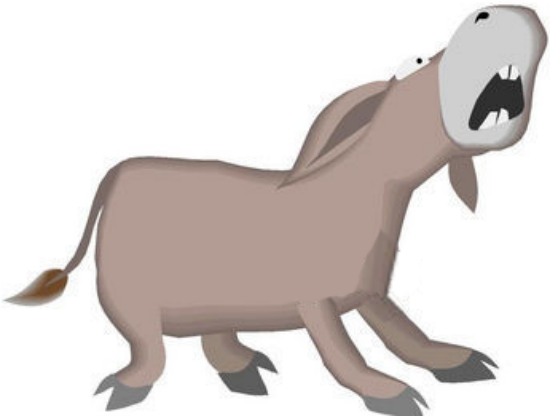
Makin' Copies

- What *should* happen when a pointer is copied? There's no "right" answer to this - it depends on what you want your code to do:
  - only let one pointer point to an object
  - create a whole new copy of the object pointed to
  - transfer "ownership" when a pointer is copied
  - use reference counting
  - use reference *linking*: maintain a list of pointers that point to the same object
  - use copy-on-write

Dangling Pointers

- auto_ptr uses the first strategy: only allow one pointer to point to an object
- How does the following code accomplish this?

```
template <class T>
auto_ptr<T>& auto_ptr<T>::operator=(auto_ptr<T>& rhs)
{
    if (this != &rhs) {
        delete ptr;
        ptr = rhs.ptr;
        rhs.ptr = NULL;
    }
    return *this;
}
```



Copy-On-Write

- For big objects, something called copy-on-write works very well
- With COW, only the pointer is copied...
- ... but when before the object is modified, a separate copy of the object is made and the new copy is what gets changed
- This gives the illusion of each pointer "owning" its own object, but can be more efficient
- How can COW make things more complex?

Exception Safety

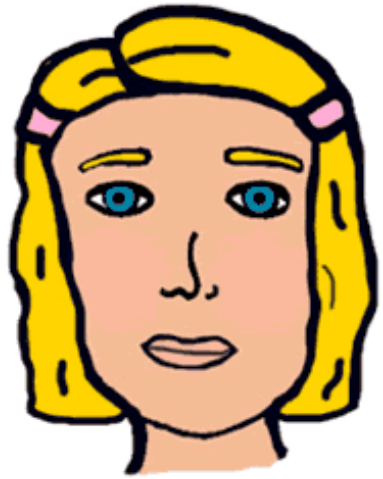
- Here's a simple code snippet using pointers:
- What happens if DoSomething() throws an exception?
- How would smart pointers help this situation?

```
void foo()
{
    MyClass* p(new MyClass);
    p->DoSomething();
    delete p;
}
```

STL Containers

- The easiest way to store different kinds of objects in a single container object
  - Like in Project 6: make a **vector<Account*>** and we can store pointers to any object derived from Account in that vector
- The big problem with this: cleanup!
- After we're done with the container we need to manually delete all the pointers in there
- Also, this is not exception-safe!

More STL-ness



- How would we use **auto_ptr**'s to deal with these problems?
- How would that help?
- Random note: STL containers sometimes copy/delete their elements behind the scenes. This means that some smart pointers are *not* safe to use with STL containers. Why not?

Which Kind 'O Pointer?

- **local variables:** auto_ptr is usually the right choice
  - it's simple, it's standard
- **class members:** you *could* use auto_ptr... why would this not always be a good choice?
- **STL containers:** auto_ptr has the previously-discussed issues; generally another class is needed