C++: Memory Problems

or
When Good Memory Goes Bad

Logistics

- Project
  - Part 2 (water) due Sunday, Oct 16th
- Questions?

Logistics

- Final exam
  - Good news…bad news
  - Good news
    - Last day of finals, November 18th
  - Bad news
    - 8am-10am
  - Room
    - 01-3338

Plan for this week

- Today: Intro to Memory Management
- Tomorrow: Memory Problems
- Thursday: Standard Libraries

Memory Leak

- A bug in a program that prevents it from freeing up memory that it no longer needs.
- As a result, the program grabs more and more memory until it finally crashes because there is no more memory left.
- In short:
  - Allocating without cleaning up.

Memory Leak

```cpp
class Foo {
    private:
        int *array_member;
        int asize;
    ...
    public:
        Foo (int size);
        ~Foo ();
};
```
Memory Leak

```cpp
Foo::Foo (int size) :
    asize (size), array_member (new int[size])
    // size, array_member = new int[size];
    { for (int i=0; i<asize; i++)
        array_member[i] = 0;
    }
void f ()
    // local aClass object
    aClass a (20);
    ...
```

Pointers Ownership

- Everything that is a pointer should be owned
  - Responsible for cleanup when finished
  - Should be known to programmer
  - Should be by design during implementation.

  - Owner and only owner should perform a delete.

Pointer Ownership

```cpp
// constructor
Foo::Foo (int size) :
    asize (size), array_member (new int[size])
    // size, array_member = new int[size];
    { for (int i=0; i<asize; i++)
        array_member[i] = 0;
    }
  // destructor
Foo::~Foo ()
    { delete [] array_member;
    }
```

Pointer Ownership

```cpp
// copy constructor
Foo::Foo (const Foo &F)
{ if (F != (*this) { // F != this
    delete [] array_member;
    array_member = new int[F.asize];
    asize = F.asize;
    for (int i=0; i<asize; i++)
        array_member[i] = F.array_member[i];
    } }
```
// assignment operator
Foo &Foo::operator= (const Foo &F)
{
    if (F != (*this)) {
        delete [] array_member;
        array_member = new int[F.asize];
        asize = F.asize;
        for (int i=0; i<asize; i++)
            array_member[i] = F.array_member[i];
    }
    return (*this);
}
Pointer Ownership

Memory is static… object should be responsible for pointer but no deallocation necessary

char * Moo::getID()
{
    // This is okay too.
    static char idInFunct[50] = "I am a cow";
    return idInFunct;
}

Pointer Ownership

Should not return pointer to local variable

char * Moo::getID()
{
    // This is not okay.
    char idInFunct[50] = "I am a cow";
    return idInFunct;
}

Pointer Ownership

• Pointers returned by functions
  – Who should be responsible for memory to which these pointers point?
    • Either caller or object
    • Should be clearly designed and documented

Pointer Ownership

• Anonymous Objects
  – An anonymous object is an object in every sense except that it has no name.
  – Used for creating very temporary objects.

Point square[] =
    {Point(0,0),Point(0,1),Point(1,1),Point(1,0)};

Pointer Ownership

• Anonymous Objects
  – Beware when anonymous objects are allocated on free store.
    vector< Card * > hand;
    hand.push_back( new Card(...) );
    hand.push_back( new Card(...) );
    hand.push_back( new Card(...) );
    hand.push_back( new Card(...) );

    If vector does not take ownership of the objects stored in it, a memory leak is possible.

Memory Leak / Pointer Ownership

• Questions?
Dangling Pointers

- Pointer is pointing to something that it shouldn’t be.
- Can happen if:
  - If the scope of a pointer extends beyond that of the object being pointed to
    - i.e., returning a pointer to a local variable.
  - If a dynamically allocated memory cell is freed explicitly and then the pointer pointing to such a space is used in subsequent code.

```cpp
Dangling Pointers
p1 = new Foo;
delete p1;
p2 = new Bar; // What if same memory is
given to p2?
int i = p1->data; // i contains garbage
p1->op(...); // p2's object mysteriously changes!
```

Dangling Pointers

- Ways to prevent dangling pointers
  - Do not return pointers to local variables.
  - After calling `delete` on a pointer, immediately set it to NULL.

```cpp
Dangling Pointers
p1 = new Foo;
delete p1;
p1 = 0;
p2 = new Bar;
p1->op(...); // core dump!
```

Dangling Pointers / Double delete

```cpp
Dangling Pointer / Double delete
class Foo
{
private:
  int *array_member;
  int asize;
  ...
}
void moo ()
{
  Foo c1, c2;
  c1 = c2;
}
Aka premature deletion
```

Bounds errors

- Recall that C++ has no array bounds checking.
  - Writing past bounds of array will trash unsuspecting memory.

```cpp
Bounds errors
void foo()
{
  int *a = new int[20];
aClass *b = new aClass();
...
a[20] = 23; // b mysteriously changes
}
```

Bounds errors

```cpp
Bounds errors
Free store
heap
```
Bounds errors

- This can be quite dangerous for local arrays:

```c
void foo()
{
    int a[20];
    a[20] = 23;
}
```

Dangling Pointers / Bounds errors

- Questions?

Getting around these problems

- The smart pointer
  - Prevents memory leaks and dangling pointers
  - Wrapper class that owns a pointer to an object
    - Object keeps a reference count of variables accessing it
    - When the reference count reaches 0, the object is deleted by the smart pointer.
    - After deleting object, pointer value set to 0.
The Smart Pointer

- The smart pointer should look and act like a regular pointer:
  - Should support ->
  - Should support *
- Should be smarter than your average pointer.
  - Reduce memory leaks
  - Reduce dangling pointers
  - Take ownership

Using smart pointers

Instead of

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

Use

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

`p` will cleanup after itself

Smart Pointers

- We may be looking at the details of a smart pointer in Week 9/10
- There is a Smart Pointer class available for use on the project
  - See choices.html of Project pages
- Questions?

Summary

- Memory Woes
  - Memory leak
  - Dangling Pointer
    - Double Delete (aka Premature delete)
  - Pointer Ownership
  - Overwrite array

Tomorrow’s Lab

- Inheritance

- Memory Management Lab in 2 weeks.