Memory Strategy in Macintosh Programs

Darin Adler
Bent Spoon Software

MacHack, June 1999

Memory questions.

- Speed of allocation.
- Memory exhaustion.
- Memory leaks.
- Locality of reference.

I’ll be covering memory exhaustion today.

- You’ll see that there’s enough in that single topic for a long talk and we won’t even cover enough of that topic.
- I’ll focus on C++ techniques. Many are applicable to other languages.

What do we want to happen when out of memory?

- Program terminates?
- Report that the operation failed?
- Tell the user to close documents?

What happens if we ignore the issue?

- With C’s malloc, we get a pointer that points to address 0.
- With NewPtr, we get a pointer that points to address 0.
- With NewHandle, we get a handle that points to address 0.

What happens if we ignore the issue?

- With modern C++'s new, the program terminates with an uncaught exception.
- With some Macintosh toolbox routines, we get unpredictable behavior.

Why can programmers get away with wishful thinking?

- On platforms other than Macintosh, they may be able to ignore the issue because of how virtual memory works.
- Perhaps because the machine is guaranteed to get slower as you allocate more memory.

Like trying to travel at the speed of light.

- The closer you get to running out, the slower the computer gets.
- Can never run out just like you can’t travel at c.
- Perhaps we need a new slogan.

Our new slogan.

- We are Macintosh programmers:
  We can travel at the speed of light!

(I guess it’s not really a good thing.)

Worst cases.

- On UNIX, termination is usually OK.
- C programs often terminate with a core dump trying to access address 0.
- C++ programs often terminate with an uncaught exception.

Survey of techniques.

- Detecting memory exhaustion.
- Handling memory exhaustion.
- Reporting to the user.
- Strategies.

Detecting memory exhaustion.

- Consider the interface of each operation that can consume memory.
- Choose between:
  - Error code returned by the call.
  - Signal value to indicate the call failed.
  - Exception thrown by the call.

A bigger problem.

- Many Macintosh toolbox routines will malfunction if there isn’t enough memory.
- Some kind of protection is necessary.

Handling memory exhaustion.

- Everybody reports memory exhaustion, but no one wants to do anything about it.
- It’s common to ignore error codes, signal values, and exceptions.
- The language doesn’t help; it’s hard to document the error behavior.

Error behavior definition.

- Document which errors can arise.
- Distinguish errors that can happen at runtime unpredictably and those that can only happen because of a programming mistake.
- If an error occurs, does the routine undo any partial operation?

Notice memory exhaustion before it's too late.

- A common technique for this is the memory reserve.
- Another technique is preflighting.

Memory reserve.

- This is memory that's kept around so that it can be released when otherwise out of memory.
- The reserve typically must be implemented as an actual block that is allocated and freed.

Strategies.

- Use exceptions, because it's easier to get them right than error codes.
- Define most routines so they do nothing permanent if they fail.
- Use memory reserves.

Can not checking be a viable strategy?

- Surprisingly, yes, at least for some.
- This requires knowledge of the maximum your program will need, and can be trickier than checking.
- But it does result in an smaller, apparently simpler program.
- I’ll discuss the checking strategy.

Code bloat.

- Some say that adding memory checks to their routines will make their code too big.
- When using exceptions, it’s important to make covers that throw the exception instead of constantly turning error codes into exceptions.
  - I think PowerPlant gets this wrong.

Code bloat.

- There is a code size increase caused by the additional code paths generated by code that can throw exceptions.
- The source code can bloat up if you have a lot of try/catch code.

A cover routine.

- Here’s an example of a cover routine.

```
Handle newHandle(Size size)
{
    Handle result(NewHandle(size));
    if (result == NULL)
        throw std::bad_alloc();
    return result;
}
```

Getting exceptions right.

- It may be easier than error codes, but it’s still tough.
- The main issue is cleanup at all the possible points where an exception might be raised.

Avoid catch(...) for cleanup.

- Destructors work much better.
- With catch(...), it's easy to get it wrong and think it's right.
- Create your own simple classes so destructors can clean up, or start with the ones from PowerPlant.

Use an object with a destructor.

- Examples.
  - Changing CurResFile().
  - Allocating memory.
  - Creating a file.
  - Opening a file.
- Once you get started, it’s fun.

How to deallocate memory automatically.

- Use vector<char>.

```
std::vector<char> v(count);
ReadData(&v[0], count);
```

- Difficult to do this with auto_ptr.
- Or use PowerPlant’s StPointerBlock.

Write destructors carefully.

- Destructors should not throw exceptions.
- An exception thrown during exception handling causes the program to terminate.

Use reserves to handle the three big problems.

- Guarantee to run the user interface when otherwise out of memory.
- Avoid exercising worst-case memory handling in your program in the field.
- Protect toolbox routines so they aren’t called with insufficient memory.

User interface reserve.

- Use this so when you are out of memory, the user can free some.
- When this reserve is gone, do not allow high-level user actions that result in more allocation.

Naughty calls reserve.

- Use this so you won’t call the toolbox with insufficient memory.
- Before calling each “suspect” function, check the reserve.
- If the reserve is not available, fail as if you had exhausted memory.

Soft failures reserve.

- Use this so that actual exercise of memory exhaustion code is reduced.
- This should only be in production versions of your program. Test without it as much as possible.
- Bad luck is still an issue, but reduces the probability of “tests” in the field.

Releasing reserves.

- The user interface reserve is released in the main event loop if the free memory gets below a threshold.
- The soft failures reserve is released if memory is exhausted, in a new handler or grow zone function.

Releasing reserves.

- The naughty calls reserve is released if memory is exhausted and the soft failures reserve is already gone.

Behavior when reserves are missing.

- When the user interface reserve is not available, the commands in the main event loop are reduced to a set that will not allocate additional memory.
  - Can’t open a new document.
  - Can close an existing document.

Behavior when reserves are missing.

- If naughty calls reserve is missing, code that runs before each naughty call will throw an exception rather than calling through.
- If soft failure reserve is missing, user interface reserve is released since you are below the threshold.

Reallocating reserves.

- Each time through the event loop, we reallocate the soft failure reserve.
- If we have the soft failure reserve, each time through the event loop we reallocate the user interface reserve.
  - Tell the user when we get that one back.

Reallocating reserves.

- We reallocate the naughty calls reserve just as we check it, before making a naughty call. If we don't manage to reallocate it, then the call fails as mentioned before.

Reserve sizes.

- User interface reserve is big enough to run the event loop and UI.
- Naughty calls reserve is the largest amount of memory that is used by the most memory-hungry naughty call.
- Soft memory failures reserve is bigger if you want more errors to be soft.

How close is PowerPlant to this strategy already?

- PowerPlant has the user interface reserve, but doesn’t have the others.
- There’s no guarantee that there’s enough memory to keep the program running once the reserve is gone.

Testing.

- There are more code paths than you think with exceptions.
- Every function that can raise an exception creates another code path that must be tested.
- The only way is to simulate the error condition.

Not the end of the matter.

- Use those exceptions.
- Use reserves or the equivalent to handle the big three problems.
- Come up with a better strategy and teach it to me.

Discussion, questions.