

Lecture Topics

- overloading new and delete
- profiling and tuning: measuring operation timing

Administrivia

- Lab #1 online yesterday afternoon; due in a week

Overloading New and Delete

- Str. Ch. 10 discusses memory management in detail
- both operators can be overloaded
 - overloading of `new` is fairly flexible
 - can overload `delete`, but not all versions can be reached
 - when overriding default memory management
 - include C++ standard header
 - `#include <new>`
- overloading operator `new`
 - `std::size_t` argument (implicit in calls)
 - must appear as first argument in all operator `new` signatures
 - holds number of bytes needed when called
 - array and instance allocation are distinct even by default
 - signatures

```
void* operator new (std::size_t size);
void* operator new[] (std::size_t size);
```
 - for a class ALPHA:

```
ALPHA* a;
a = new ALPHA (1, 2, 3); // operator new
a = new ALPHA[10];      // operator new[]
```
 - note: array allocation requires a constructor with no arguments
 - can extend either/both versions with arbitrary arguments
 - for example

```
operator new (std::size_t size, int region_id);
```
 - other arguments are then passed to `new` as follows

```
a = new(42) ALPHA (17, "potato");
```

- overloading operator `new` (cont'd)
 - one use of extra arguments: placement
 - want some control over location of “dynamic” allocation
 - e.g., sometimes necessary to locate instances in DMA-accessible memory (low physical addresses)
 - default placement
 - locate at a specific place (provided as argument to `new`)
 - `void* operator new (std::size_t size, void* p) {return p;}`

- exception handling
 - header file `<new>` also defines an exception for allocation failure
 - `std::bad_alloc`
 - derived from `std::exception`
 - versions of `new` discussed so far can generate exceptions

```
void* operator new (std::size_t size) throw (std::bad_alloc);
void* operator new[] (std::size_t size) throw (std::bad_alloc);
```

- notation implies that no other exceptions are thrown
- default versions also exist that return NULL instead
 - header file `<new>` defines structure and static variable to allow pseudo-argument

```
void* operator new (std::size_t size, const std::nothrow_t&)
    throw ();
```

- notation here implies that no exceptions are thrown
- to invoke this form, use something like


```
a = new(std::nothrow) ALPHA (42, "no exceptions");
```

- exception handling before exceptions
 - some of the C++ support pre-dates the exception-handling mechanisms
 - this support is **not thread-safe**, and you should generally avoid it

 - default operator **new** behavior
 - try to allocate
 - on failure, check whether a handler has been registered for failures
 - if so, call it and try again
 - if not, throw an exception

 - you can register a handler with

```
new_handler set_new_handler (new_handler) throw ();
```

 - **new_handler** is a pointer to a function that
 - takes no arguments
 - returns nothing (void)
 - **set_new_handler** returns the previous handler pointer

 - note that the default behavior keeps calling the handler
 - if handler can't fix the allocation problem
 - e.g., by garbage collection
 - it must throw an exception
 - to avoid an infinite loop

 - again
 - this mechanism **IS NOT THREAD-SAFE**
 - your program has one global variable for the handler pointer

- overloading operator `delete`
 - `gcc` will let you define many versions
 - but only two versions are usable
 - non-array

```
void operator delete (void* p);
delete a;
```
 - array

```
void operator delete[] (void* p);
delete[] a;
```
 - note
 - compiler **does not check** that you used the “correct” version
 - it does not remember which version of `new` you used
 - it allows either version of `delete` without warning
 - the book rambles for a while on rationale
 - tries to establish the difficulty for programmers to track “types” of allocations and use proper deallocations
 - instead, have `new` record type and `delete` make use of it
 - except that the array version allows exactly that type of oddity
 - what’s the real reason?
 - probably the fact that the following are synonymous
 - `delete(a)`
 - `delete a`
 - similar to people writing `return (42);`
 - and thus lots of code might break to support argument-passing to `delete`

- recall brainstorming question:
Given a short operation sequence, what makes it hard to measure accurately?

- some issues
 - timer overhead
 - timer granularity
 - processor/resource virtualization
 - interrupts
 - hot/cold caches, TLBs (what's the right level for a module?)
 - pipelining, cache alignment, etc.

- a common strategy...
 - avoid timer issues by measuring many, not one
 - run on otherwise empty system to avoid conflicts
 - interrupts should be small relative to cost; use minimum timing result
 - assume (and force) hot caches
 - weak argument: important and frequent op won't have cold caches
 - reality: sometimes hard to know without actual application
 - ignore pipelining