General Optimization

Optimized C++

Ed Keenan

27 October 2019

13.0.6.4.19 Mayan Long Count

Copyright 2019, Ed Keenan, all rights reserved.

DEPAUL UNIVERSITY

Goals

- Everyone is connected
- Easy Way
  - Compilers
  - Money
- Source of inefficiency
- Techniques
  - Loops
  - Logic
  - Strings

I’m Yojimbo!

General observations

- Everything is interconnected
  - Optimizing on one section, affects others
  - You might shuffle the whole system and make is slower
- Existing architecture might be the biggest challenge.
  - It’s hard to optimized when system isn’t modular or encapsulated
  - Might need to Refactor to make the system clearer

More observations

- Class design
  - Sometime the internal structure of classes and it layout dictate performance
  - Can you swap out routines easy?
  - Is their in-lining opportunities?
  - What is opaque versus observable?

Compilers

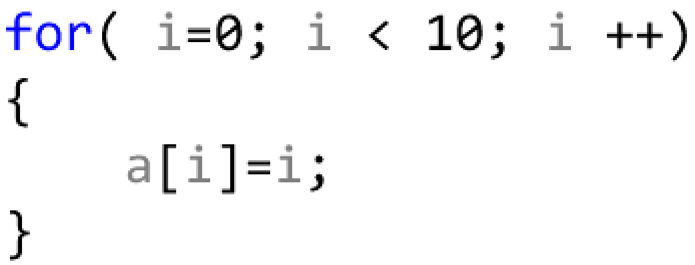
- Compilers
  - Huge difference in code performance
  - Intel creates optimal code for Intel processors.
    - How is that possible?
  - Compiler settings
    - Need to be understood and tweaked
- Compilers are smarter than you…
  - Almost, but generally better return for the dollar

Throw Money at it?

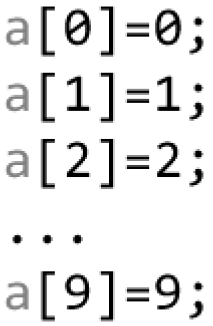
- Buy faster and better hardware
  - Is that enough?
- Why isn’t my code going twice as fast?
  - I/O, Disc, Networking might be a bottleneck
  - Algorithms don’t scale
  - Many processors, code only uses one.
- New systems have more but slower procs
  - Cost savings to manufacture
  - IBM servers…

Line Count?

- Reducing the lines of code, improves speed
  - Many things are happening under the hood.
  - Which is faster?



```
for( i=0; i < 10; i ++)
{
    a[i]=i;
}
```



```
a[0]=0;
a[1]=1;
a[2]=2;
...
a[9]=9;
```

Understand functions

- All operations are not created equally
  - Operating system calls, like copy, read, sort take a very long time.
  - Copy constructors, passing by value are deceiving
- Optimize everything
  - You have limited time and money to work
  - Spend them where it counts
  - Resist premature optimization
  - Later we'll talk about Performance Solution Engineering (PSE) in Week 9

Premature Victory

- Fast program is good enough
  - My program is:
    - 90% or almost working.
    - Practically done
  - Wrong, if it’s not working it’s not optimized.
    - Its generally the edge conditions that hurt clean streamlined code.

Source of Inefficiency

- Input / Output operations
  - Biggest and most evil
    - You have to deal with it.
  - Being clever can reduce it’s effects.
    - Only use it if you have to.
  - Ways to improve it
    - Cache copies
    - Stream
    - Layout all help
    - Learn the hardware

Source of Inefficiency

- Memory
  - Yes it comes up everywhere
    - Very slow, we can do better
  - Custom schemes
    - Writing for our use cases
  - Virtual memory manager
    - Transparent but cost time
- Thread and process switching
  - Other heavy system calls
  - They hurt ☹

Source of Inefficiency

- Language choices
  - Compiler vs interpreted
    - Interpreted is roughly 20 times slower
    - There is a place and time for them
    - Not in the most call systems or loops
    - Remember the 80/20 rule
  - Maintenance vs speed
    - Support and readability sometimes get sacrificed for optimization

Source of Inefficiency

- Errors
  - Leaving debug information
  - Not freeing memory
  - Redundantly initializing memory
  - Unnecessary initializations
  - Bugs
  - Do you know what the code is really doing?
  - Many people do not step code or know how

Metrics

- Don’t be a Cowboy or Cowgirl
  - Your spidey sense isn’t that good.
  - Measure everything!
  - Assume nothing!
  - Story about Mips processor

Heisenberg Uncertainty Principle

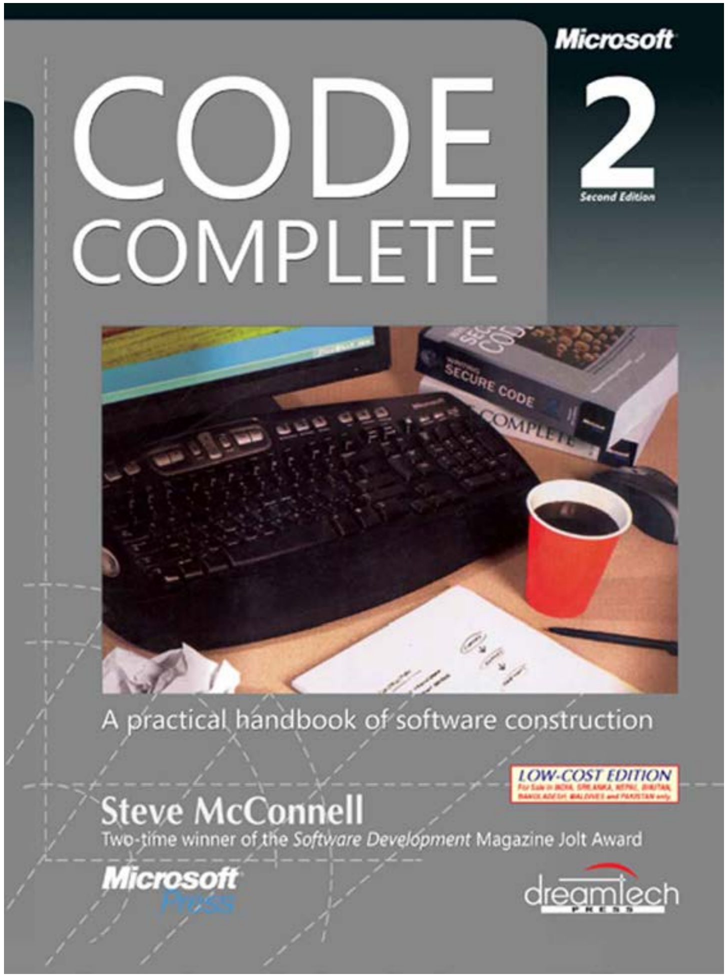
- By observing the system you affect the results of the system
  - Mexico Story
- Optimizations
  - If you alter one system,
  - The next system might be negatively altered.

Break

- What time is it?
  - I’m Thirsty!!!

Techniques

- Reference:
  - Chapters 25 & 26 of Code Complete 2nd Edition
- Good News – Safari free for DePaul
  - http://proquestcombo.safaribooksonline.com/book/software-engineering-and-development/0735619670


Microsoft
CODE COMPLETE
2
Second Edition
A practical handbook of software construction
LOW-COST EDITION
Steve McConnell
Two-time winner of the Software Development Magazine Jolt Award
Microsoft Press
dreamtech PRESS

Logic

- Conditionals
  - Early out
  - Reworking conditionals to use
- Does every know their Binary Logic?
  - And, Or, Xor, Not, Xnor,
  - Associative, Commutative, Distributive
  - DeMorgan?
    - !(x+y) = !x & !y
    - Why is this important?

DeMorgan

- Conditionals
  - Evaluation happens from left to right.
- If (x && y)
  - If x if false, no need to evaluate y
  - Early out.
- If (x||y) then …
  - If you can rework the logic to be negative
  - If !(x||y) then …Can use DeMorgan
- If (!x && !y) then …
  - You get the early out ☺

Switching / exiting

- Understand how switch() work!
  - They are implemented under the hood as
  - Many if else…
- Invariants
  - Do not have invariant states inside the loop
  - Only keep statements that change in the loop.
- Sentinals
  - What are they?
    - One time flags inside of loops
  - Evil, check is done every time
    - Move out of loop

Loops

- Combining work inside with same loops
  - Sometimes this is counter intuitive
  - Need to test, caching becomes a big issue
- Unrolling
  - Sometimes helps,
  - some times confuses compilers
- Busiest loop in the inner most loop
  - Multi-nested loops
  - Less loop-context switching

Floats

- Floats vs Integers
  - Floats are good for math
    - Not for indices or conditional testing
  - Integers are great of conditionals and indexing
    - Well kind of?
    - Should not be passed to floating point parameters
- Multi-dimensional arrays are slow
  - Indication of poor design
  - Refactor

Strings

- Strings
  - Very slow to compare and use
  - Get creative, do you really need them?
  - In industry, cause of many slowdowns
  - Often, too embedded in the existing architecture to remove ☹
- MD4 or MD5 quick alternative to strings
  - Allows integer compares
  - Fixed length strings… sound weird?
    - Making all your strings the same length has advantages in processing them.

System Calls

- Understand your system calls
  - Many take doubles, when you need floats.
    - Sqrt() is big offender.
  - Why do people need 64-bit floats when we went to the moon on 16 bit fix point?
- Bit shifts and tricks don't work anymore.
  - Look at timing and metrics
  - They are implement with slow legacy and often with slow your project down.

Assembler

- Assembler
  - Just like Copernicus
  - You can’t practically do this for the N-stage pipeline in processors, with look ahead and in order executions, cache misses, hyper-threaded context switching, and vectorize embedded coprocessors.
  - We will learn intrinsics operators
    - Like mini micro assembler functions

Thank You!

- Easy?

Memory Overloading

Optimized C++

Ed Keenan

Copyright 2019, Ed Keenan, all rights reserved.

DEPAUL UNIVERSITY

Goals

- Overloading Memory
  - C Functions
    - Malloc / Calloc / Realloc / Free
  - C++
    - New / Delete
  - STL
    - Allocators
- Useful at C++ and OO conventions
  - 1st conversation of the night

Overload?

C Functions

- Overload standard C Functions
  - Malloc
    - Memory Allocations
  - Calloc
    - Clears the memory it returns for an allocation
  - Realloc
    - Resize allocations
    - Swiss Army Knife of allocations
  - Free
    - Deallocates previous allocations

Know your Definitions

- If you overload any function
  - You must know the function’s spec inside and out.
  - Users assume same behavior
- Many programmers uses functions that they don’t understand
  - ***VERY BAD!*** ☹
  - Take 1 minute and read the Online manual
  - Good References
    - http://www.cplusplus.com/reference/clibrary/cstdlib/
    - http://msdn.microsoft.com/en-us/library/dtefa218%28v=vs.110%29.aspx

Malloc

- Allocate Memory Block
  - void * **malloc** (size_t **size**);
- Definition
  - Allocates a block of size bytes of memory, returning a pointer to the beginning of the block.
  - The content of the newly allocated block of memory is not initialized, remaining with indeterminate values.
- Parameters
  - Size
    - Size in bytes of the allocations
- Return Value
  - On success, a pointer to the memory block allocated by the function.
    - The type of this pointer is always void*, which can be cast to the desired type of data pointer in order to be dereferenceable.
    - If the function failed to allocate the requested block of memory, a ***NULL*** pointer is returned.

Calloc

- Calloc
  - Clears the memory it returns for an allocation
  - What does ***C*** – stands for?
    - ***C****lear* – clears the memory
    - ***C****ount* – uses count in its arguments
    - ***C****ontiguous*, ***C****ore*, ***C****ommit*, ***C****hunk*, and ***C****haracter?*
  - *Early K & R?*
    - **Kernighan & Ritchie – the C Programming Language**
    - *Calloc – "C" language free*
    - *Cfree – "C" language free*

Calloc

- Allocate space for array in memory
  - void * **calloc** (size_t **num**, size_t **size**);
- Definition
  - Allocates a block of memory for an array of *num* elements, each of them *size* bytes long, and initializes all its bits to zero.
  - The effective result is the allocation of an zero-initialized memory block of (*num* * *size*) bytes.
- Parameters
  - num
    - Number of elements to be allocated.
  - size
    - Size of elements.
- Return Value
  - On success, a pointer to the memory block allocated by the function.
    - The type of this pointer is always void*, which can be cast to the desired type of data pointer in order to be dereferenceable.
    - If the function failed to allocate the requested block of memory, a ***NULL*** pointer is returned.

Realloc

- Reallocate memory block
  - void * **realloc** (void * **ptr**, size_t **size**);
- Definition
  - The size of the memory block pointed to by the *ptr* parameter is changed to the *size* bytes, expanding or reducing the amount of memory available in the block.
  - The function *may move* the memory block to a new location, in which case the new location is returned.
  - The content of the memory block is preserved up to the lesser of the new and old sizes, even if the block is moved. If the new *size* is larger, the value of the newly allocated portion is indeterminate.
  - In case that *ptr* is NULL, the function behaves exactly as malloc assigning a new block of *size* bytes and returning a pointer to the beginning of it.
  - In case that the *size* is 0, the memory previously allocated in *ptr* is deallocated as if a call to free was made, and a NULL pointer is returned.

Realloc

- Parameters
  - ptr
    - Pointer to a memory block previously allocated with malloc, calloc or realloc to be reallocated.
    - If this is NULL, a new block is allocated and a pointer to it is returned by the function.
  - size
    - New size for the memory block, in bytes.
    - If it is 0 and *ptr* points to an existing block of memory, the memory block pointed by *ptr* is deallocated and a NULL pointer is returned.
- Return Value
  - A pointer to the reallocated memory block, which may be either the same as the ptr argument or a new location.
  - The type of this pointer is void*, which can be cast to the desired type of data pointer in order to be dereferenceable.
  - If the function failed to allocate the requested block of memory, a NULL pointer is returned.

Free

- Deallocate space in memory
  - void **free** (void * **ptr**);
- Definition
  - A block of memory previously allocated using a call to malloc, calloc or realloc is deallocated, making it available again for further allocations.
  - Notice that this function leaves the value of ptr unchanged, hence it still points to the same (now invalid) location, and not to the null pointer.
- Parameters
  - ptr
    - Pointer to a memory block previously allocated with malloc, calloc or realloc to be deallocated.
    - If a null pointer is passed as argument, no action occurs.
- Return Value
  - (none)

Overloading C Libs

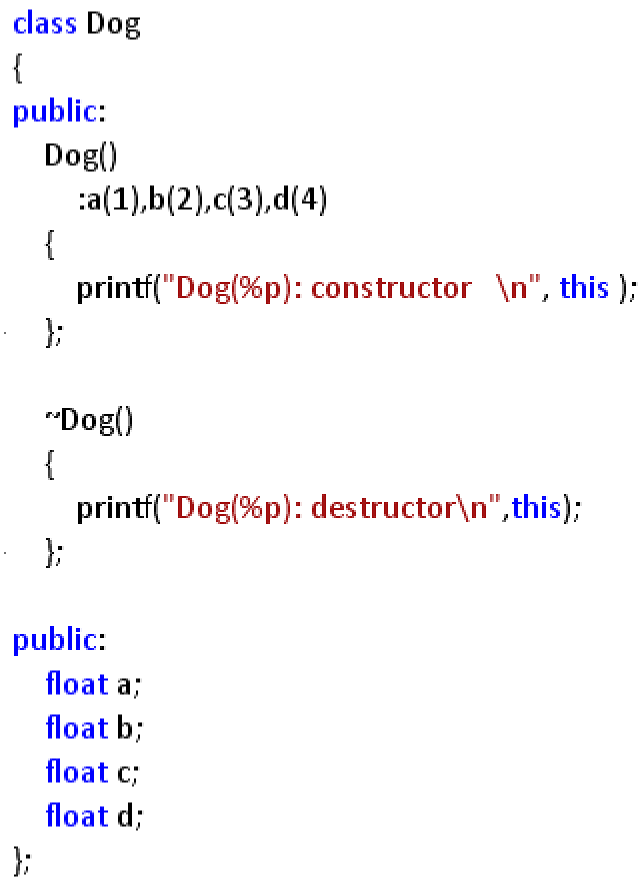
- Process

1. Create macros for your functions
   - #define **memMalloc**(size) **myMallocStdC**(size)
   - Implement macros in terms of your standard C library functions, *malloc(), calloc(), realloc(), free()*
2. Test the function replacement
   - Make sure for every malloc, calloc(), realloc() there is a corresponding free()
   - Look for every pair, If you miss one, bad stuff.
     - A mis-match will occur -> crash ☹

Cont.

3. Replace the Functions with your code
   - #define **memMalloc**(size) **myMalloc**(size)
4. Add file name and line number tracking
   - #define **memMalloc**(size) **myMalloc**(size, __FILE__, __LINE__)
5. Maintain the ability to switch back to std C
   - Useful for debug mode
     - Track performance difference
     - Track behavior or bug difference

Testbed: class Dog



```
class Dog
{
public:
   Dog()
      :a(1),b(2),c(3),d(4)
   {
      printf("Dog(%p): constructor   \n", this );
   };

   ~Dog()
   {
      printf("Dog(%p): destructor\n",this);
   };

public:
   float a;
   float b;
   float c;
   float d;
};
```

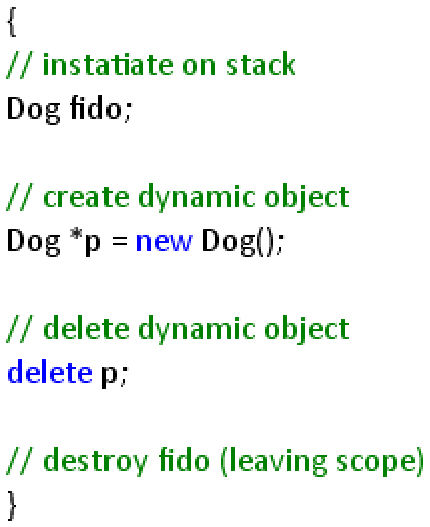
- Simple class
  - Has a few variables
  - Print statements in constructor and destructor

Dog Class
How original

Test calls

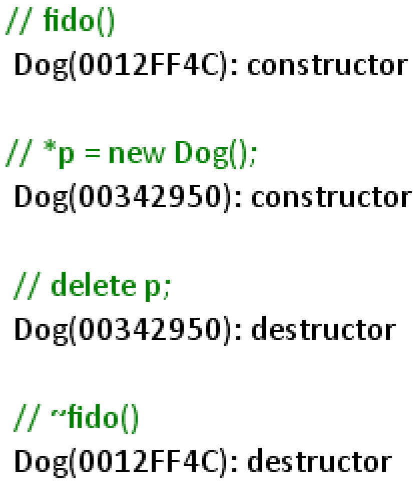
- ***fido***
  - Creation
    - Instantiated on stack
    - Constructor is called
  - Destruction
    - Leaves scope
    - Destructor is called
- ***p*** – dynamic object
  - Creation
    - Call global new
    - Constructor is called
  - Destruction
    - Call global delete
    - Destructor is called

Code:



```
{
// instatiate on stack
Dog fido;

// create dynamic object
Dog *p = new Dog();

// delete dynamic object
delete p;

// destroy fido (leaving scope)
}
```

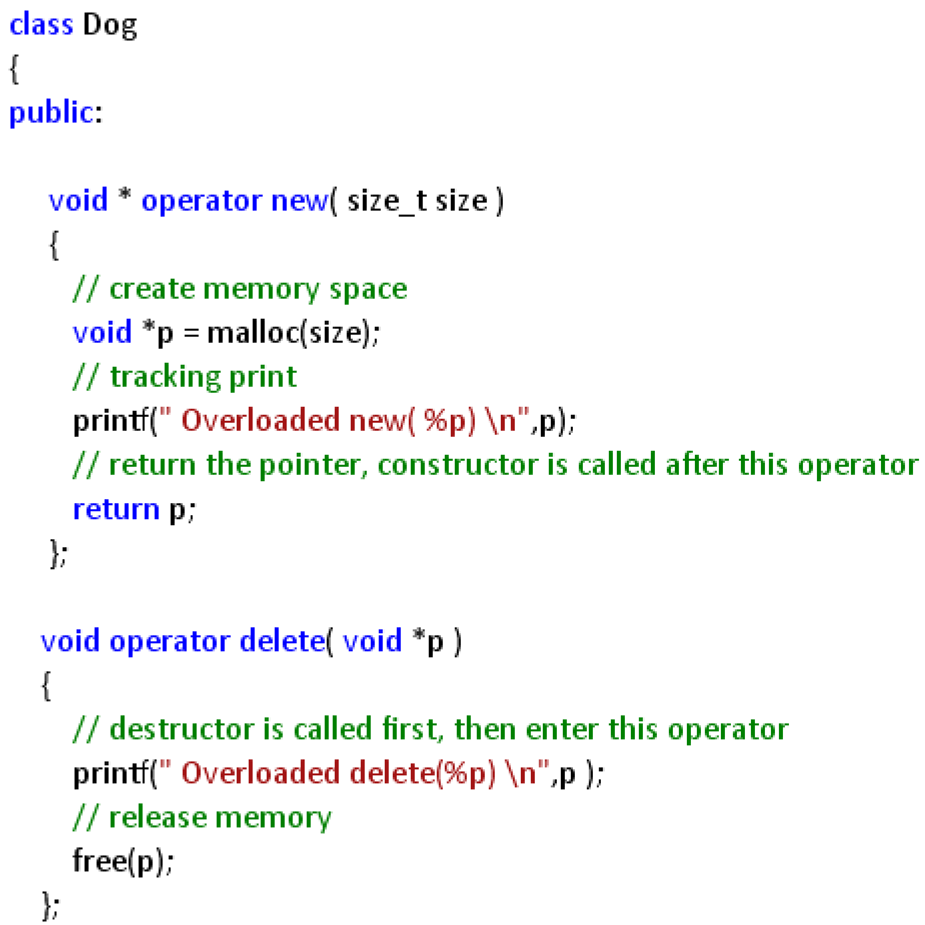
Output:



```
// fido()
 Dog(0012FF4C): constructor

// *p = new Dog();
 Dog(00342950): constructor

 // delete p;
 Dog(00342950): destructor

 // ~fido()
 Dog(0012FF4C): destructor
```

Overload in Class

- class Dog
  - Overload new operator
  - Overload delete operator
- operator **new**()
  - Need allocation
  - Allocate memory: malloc(), ***CUSTOM*** function, global new()
  - Constructor is fired off after new operator
  - Calling function doesn’t change
- operator **delete**()
  - Destructor is called before this operator
    - Automagic!
  - Release memory: free(), ***CUSTOM*** function, global delete()



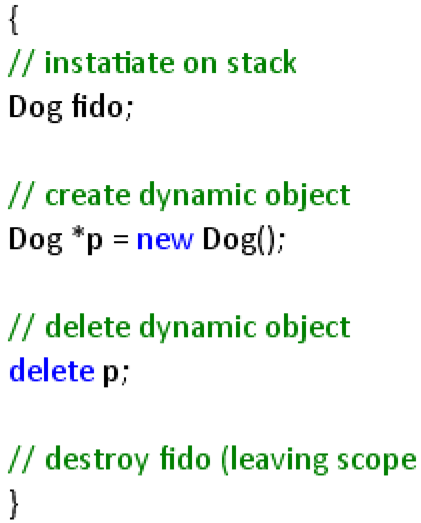
```
class Dog
{
public:

    void * operator new( size_t size )
    {
      // create memory space
      void *p = malloc(size);
      // tracking print
      printf(" Overloaded new( %p) \n",p);
      // return the pointer, constructor is called after this operator
      return p;
    };

   void operator delete( void *p )
   {
      // destructor is called first, then enter this operator
      printf(" Overloaded delete(%p) \n",p );
      // release memory
      free(p);
   };
```

DePaul University

Sample

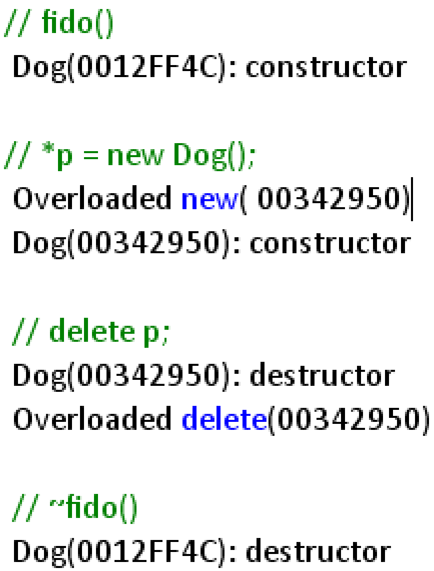
- Transparent
  - Calling function doesn't change
    - Use new the same way as before
    - Instead it calls the overloaded new of the class
  - Only overloads for that specific class
  - Make sure that allocation and deallocation match

Code:



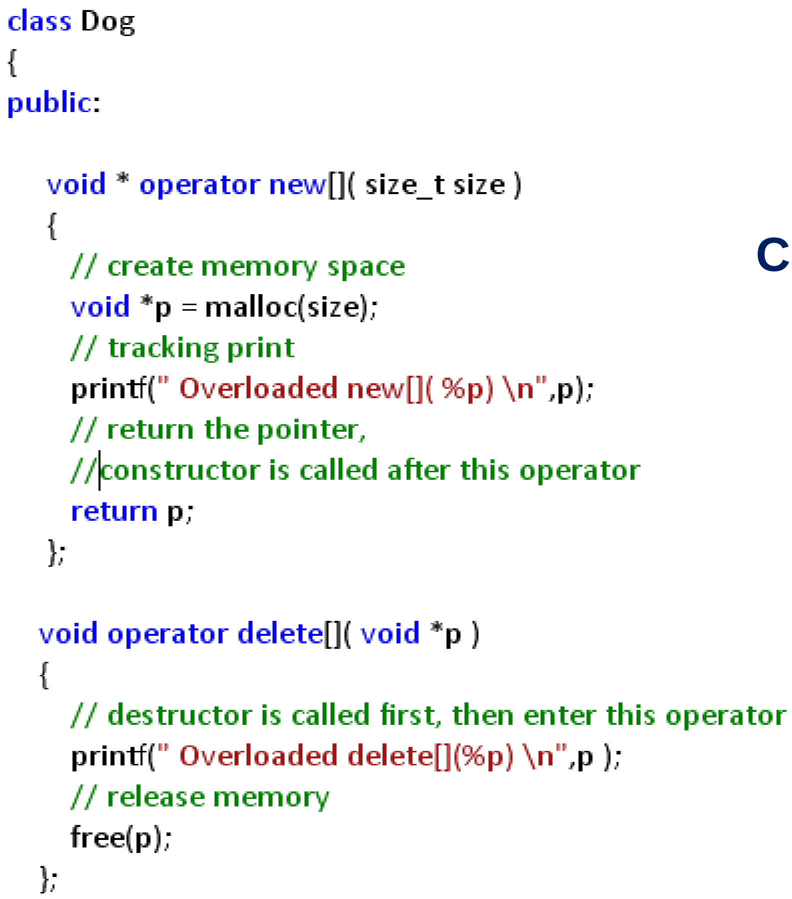
```
{
// instatiate on stack
Dog fido;

// create dynamic object
Dog *p = new Dog();

// delete dynamic object
delete p;

// destroy fido (leaving scope
}
```

Output:



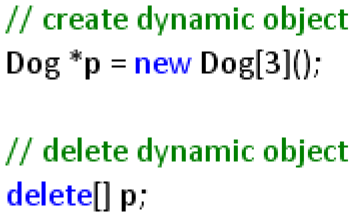
```
// fido()
 Dog(0012FF4C): constructor

// *p = new Dog();
 Overloaded new( 00342950)
 Dog(00342950): constructor

 // delete p;
 Dog(00342950): destructor
 Overloaded delete(00342950)

 // ~fido()
 Dog(0012FF4C): destructor
```

Same for [ ] operators



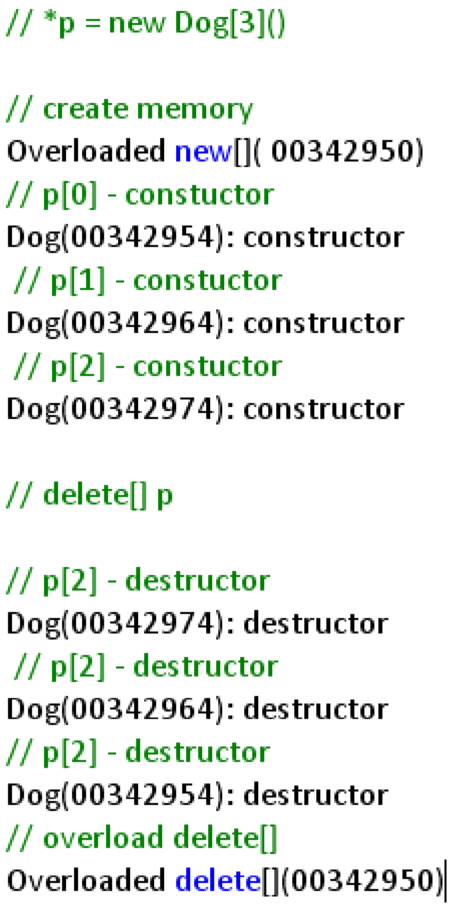
```
class Dog
{
public:

    void * operator new[]( size_t size )
    {
      // create memory space
      void *p = malloc(size);
      // tracking print
      printf(" Overloaded new[]( %p) \n",p);
      // return the pointer,
      //constructor is called after this operator
      return p;
    };

   void operator delete[]( void *p )
   {
      // destructor is called first, then enter this operator
      printf(" Overloaded delete[](%p) \n",p );
      // release memory
      free(p);
   };
```

Code:



```
// create dynamic object
Dog *p = new Dog[3]();

// delete dynamic object
delete[] p;
```

Output:



```
// *p = new Dog[3]()

// create memory
Overloaded new[]( 00342950)
// p[0] - constuctor
Dog(00342954): constructor
 // p[1] - constuctor
Dog(00342964): constructor
 // p[2] - constuctor
Dog(00342974): constructor

// delete[] p

// p[2] - destructor
Dog(00342974): destructor
 // p[2] - destructor
Dog(00342964): destructor
// p[2] - destructor
Dog(00342954): destructor
// overload delete[]
Overloaded delete[](00342950)
```

Extra info

- Useful for tracking allocations
  - Can call global new and delete in functions
    - void *p = (void *)::operator **new**(size);
    - ::operator **delete**(p);
    - Add tracking information
  - Behavior doesn’t change
- Use custom memory systems
  - This is an easy way to hook up your custom memory systems

Hazards

- Overloading global new
  - issues
- Which new is used?
  - Custom or Libraries?
    - Link order issues
    - Dynamic memory before main() is called
      - Yes it happens…

Hazards

- Fun, fun, fun… ***NOT!***
  - Get very familiar with the linker
    - Up it’s verbose warning dialog mode
  - Multiple functions defined
    - Common warnings thrown by linker
  - Control function overloading by controlling link order
    - Easy on some compilers
    - Ridiculously hard on others

Alternatives to Global New

- Create a macro
  - Include in all class declarations
  - Easy to #if out if desired
- Create a memory class
  - Contains overloaded new & delete
  - Publicly inherit to all class declarations

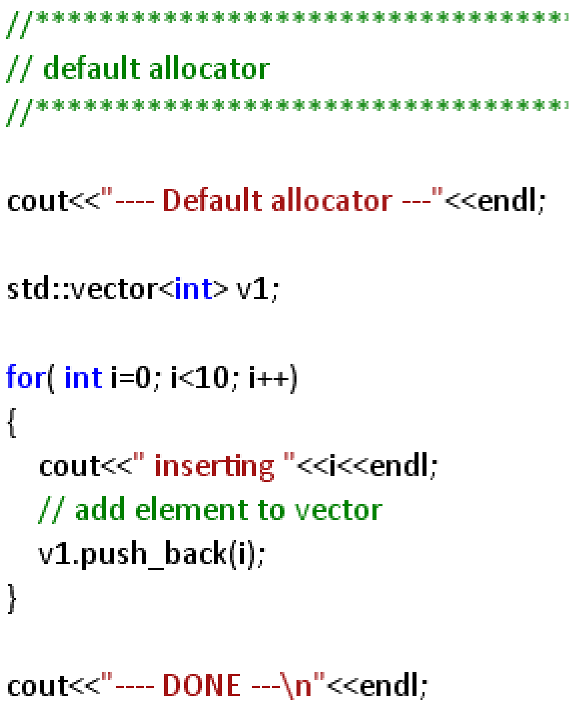
Alternatives to Global New

- Change the calling convention
  - Create your own version of new
    - #define **memNew**(heap) **new**(heap)
  - Add parameters and overload new
    - To extend and change the signature of the function.
    - To avoid collision with global new

STL allocators

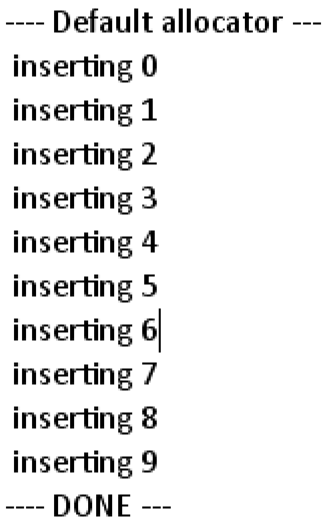
- Did you know that STL uses memory?
  - I hope so…
  - You can overload the allocator
- Why is this useful?
  - Use your custom allocator
  - Track memory allocations
  - Be a coolest power programmer in the room
    - Be a code ninja!

Example 1: default allocator



```
//*********************************
// default allocator
//*********************************

cout<<"---- Default allocator ---"<<endl;

std::vector<int> v1;

for( int i=0; i<10; i++)
{
   cout<<" inserting "<<i<<endl;
   // add element to vector
   v1.push_back(i);
}

cout<<"---- DONE ---\n"<<endl;
```

Output

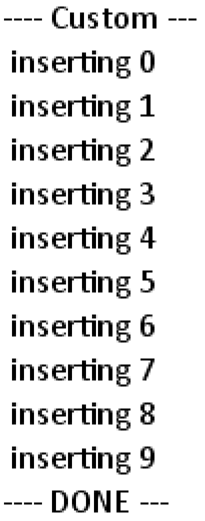


```
---- Default allocator ---
 inserting 0
 inserting 1
 inserting 2
 inserting 3
 inserting 4
 inserting 5
 inserting 6
 inserting 7
 inserting 8
 inserting 9
---- DONE ---
```

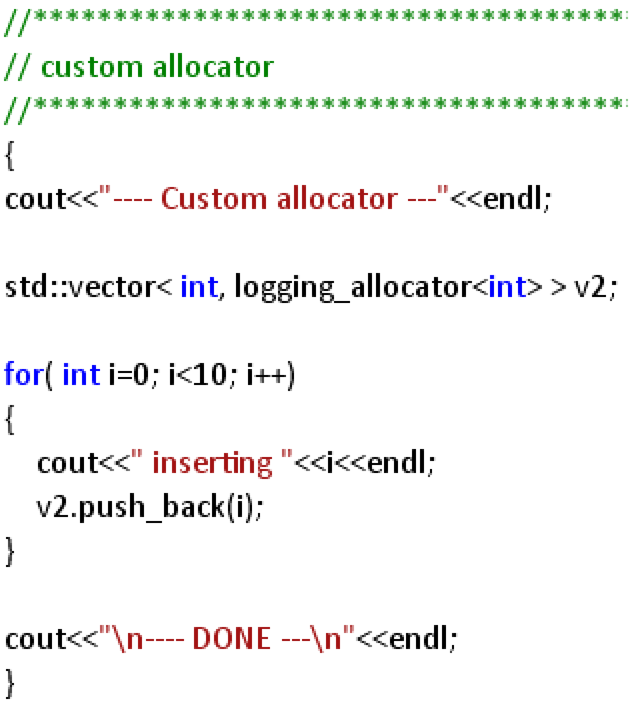
- Calls global new/delete for allocation
- What’s happening?
  - How many allocations and deletions are there?
  - How many times are the data structures copied?

Example 2: custom allocator

Console



```
---- Custom ---
 inserting 0
 inserting 1
 inserting 2
 inserting 3
 inserting 4
 inserting 5
 inserting 6
 inserting 7
 inserting 8
 inserting 9
---- DONE ---
```



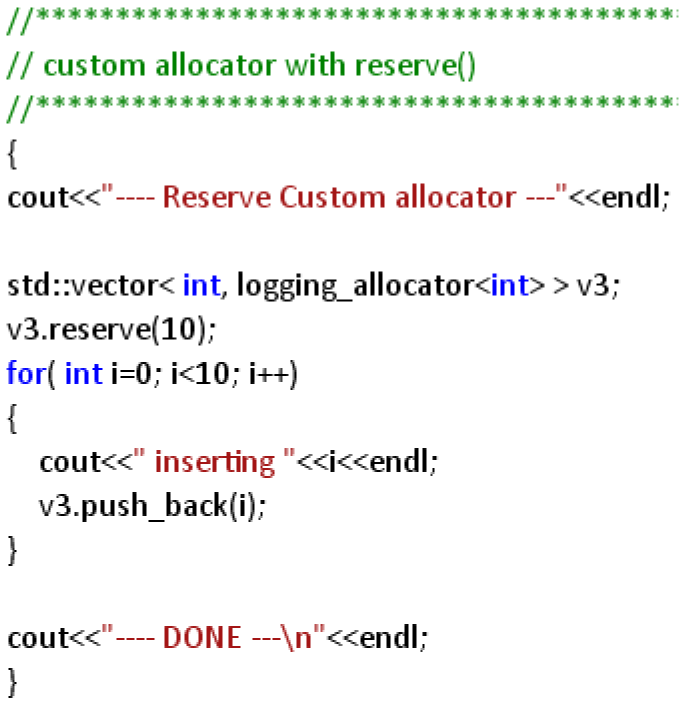
```
//************************************
// custom allocator
//************************************
{
cout<<"---- Custom allocator ---"<<endl;

std::vector< int, logging_allocator<int> > v2;

for( int i=0; i<10; i++)
{
   cout<<" inserting "<<i<<endl;
   v2.push_back(i);
}

cout<<"\n---- DONE ---\n"<<endl;
}
```

Output

```
1.   allocate() addr: 0x003B8328  num: 1  totalSize: 4
2.   allocate() addr: 0x003B95B0  num: 2  totalSize: 8
3.   dealloc() addr: 0x003B8328
4.   allocate() addr: 0x003B8328  num: 3  totalSize: 12
5.   dealloc() addr: 0x003B95B0
6.   allocate() addr: 0x003B95B0  num: 4  totalSize: 16
7.   dealloc() addr: 0x003B8328
8.   allocate() addr: 0x003B8328  num: 6  totalSize: 24
9.   dealloca() addr: 0x003B95B0
10.  allocate() addr: 0x003B95B0  num: 9  totalSize: 36
11.  dealloc() addr: 0x003B8328
12.  allocate() addr: 0x003B9610  num: 13  totalSize: 52
13.  dealloc() addr: 0x003B95B0
14.  // leaving scope
15.  dealloc() addr: 0x003B9610
```

- 7 allocations and deletions
- 6 separate copies of elements

Example 3: with Reserve



```
//****************************************
// custom allocator with reserve()
//****************************************
{
cout<<"---- Reserve Custom allocator ---"<<endl;

std::vector< int, logging_allocator<int> > v3;
v3.reserve(10);
for( int i=0; i<10; i++)
{
   cout<<" inserting "<<i<<endl;
   v3.push_back(i);
}

cout<<"---- DONE ---\n"<<endl;
}
```

Output

1. allocate() addr: 0x003B95B0 num: 10 totalSize: 40
2. dealloc() addr: 0x003B95B0

- Using Reserve saved 6 allocations and deletions
- Easy to track and monitor

Use Allocators:
To Understand STL containers

```
std::vector< data, logging_allocator<data> > v2;

for( int i=0; i<30; i++)
{
        fprintf(io::getHandle(), "inserting by push back: %d\n", i);
        v2.push_back(i);
}
```

```
inserting by push back: 0
  allocate() addr 0x003B9270  num: 1
inserting by push back: 1
  allocate() addr 0x003B92B0  num: 2
  deallocate() addr 0x003B9270
inserting by push back: 2
  allocate() addr 0x003B92F8  num: 3
  deallocate() addr 0x003B92B0
inserting by push back: 3
  allocate() addr 0x003B9270  num: 4
  deallocate() addr 0x003B92F8
inserting by push back: 4
  allocate() addr 0x003B92C0  num: 6
  deallocate() addr 0x003B9270
inserting by push back: 5
inserting by push back: 6
  allocate() addr 0x003B9318  num: 9
  deallocate() addr 0x003B92C0
inserting by push back: 7
inserting by push back: 8
inserting by push back: 9
  allocate() addr 0x003B9270  num: 13
  deallocate() addr 0x003B9318
inserting by push back: 10
inserting by push back: 11
inserting by push back: 12
inserting by push back: 13
  allocate() addr 0x003B92E0  num: 19
  deallocate() addr 0x003B9270
inserting by push back: 14
inserting by push back: 15
inserting by push back: 16
inserting by push back: 17
inserting by push back: 18
inserting by push back: 19
  allocate() addr 0x003B9368  num: 28
  deallocate() addr 0x003B92E0
inserting by push back: 20
inserting by push back: 21
```

v2.size(): 22

v2.capacity(): 28

STL Methods

Starting with a 10 element vector:

// understanding -> v.clear()

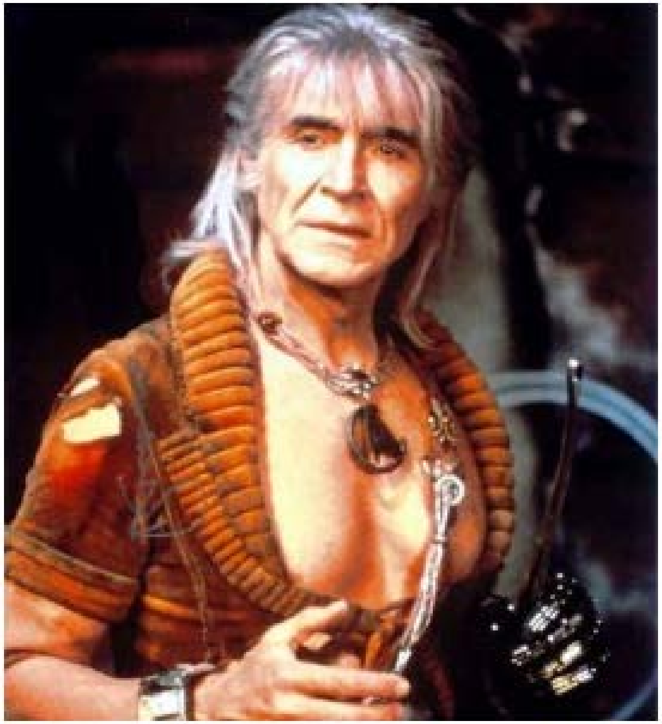
```
v2.size(): 10
v2.capacity(): 13
v2.clear()

~data(02D62C90) - destructor
~data(02D62C94) - destructor
~data(02D62C98) - destructor
~data(02D62C9C) - destructor
~data(02D62CA0) - destructor
~data(02D62CA4) - destructor
~data(02D62CA8) - destructor
~data(02D62CAC) - destructor
~data(02D62CB0) - destructor
~data(02D62CB4) - destructor
dealloc(0x0013F54C) addr 0x02D62C90  num: 13

v2.size(): 0
v2.capacity(): 0     now its in 13 C++11 spec…
```

Starting with a 10 element vector:

// understanding -> v.erase()

```
v2.size(): 10
v2.capacity(): 13
v2.erase()

~data(02D62C90) - destructor
~data(02D62C94) - destructor
~data(02D62C98) - destructor
~data(02D62C9C) - destructor
~data(02D62CA0) - destructor
~data(02D62CA4) - destructor
~data(02D62CA8) - destructor
~data(02D62CAC) - destructor
~data(02D62CB0) - destructor
~data(02D62CB4) – destructor

v2.size(): 0
v2.capacity(): 13
```

Start Trek 2: Wrath of Khan



- The gold standard (10/10)
  - All movies are compared to this movie
    - Gone with the Wind
    - Lawrence of Arabia
- Star Wars: Empire Strikes back
    - 6/10 on Khan scale.
- Matrix:
    - 7/10 on the Khan scale
- Wizard of Oz
    - 7/10 (flying blue monkeys)

Allocator

- Uses _Allocate() function to allocate memory.
  - This function calls new
  - Can be replace with your custom allocation
- STL allocation overloading is 9/10 on the Geek scale
  - Very few people have done it.
  - Ask around.

STL Allocator – Wrath of Khan

```
using namespace std;

template< typename T, typename Allocator = allocator<T> >
class logging_allocator
{
private:
		Allocator alloc;

public:
	typedef typename Allocator::size_type			size_type;
	typedef typename Allocator::difference_type		difference_type;
	typedef typename Allocator::pointer			pointer;
	typedef typename Allocator::const_pointer		const_pointer;
	typedef typename Allocator::reference			reference;
	typedef typename Allocator::const_reference		const_reference;
	typedef typename Allocator::value_type			value_type;

	// magic - you need the rebind....
	template <typename U> struct rebind
	{
		typedef logging_allocator<U, typename Allocator::template rebind<U>::other > other;
	};
```

Khan part 2/3

```
// default
logging_allocator()
{               };

// copy constructor
logging_allocator( const logging_allocator &x)
                : alloc(x.alloc)
{               };

// overloaded constructor
template <typename U>
                logging_allocator( const logging_allocator<U,
typename Allocator::template rebind<U>::other> &x)
                :alloc(x.alloc) {};

// destructor
~logging_allocator(){};

// return the address of the allocation
pointer address( reference x ) const
{
                return alloc.address(x);
};

// return the constant address to the allocation
const_pointer address(const_reference x) const
{
                return alloc.address(x);
};
```

```
// return the max size
size_type max_size() const
{
                return alloc.max_size();
}

// effectively operator new
void construct( pointer p, const value_type &val)
{
                alloc.construct(p, val);
}

// effectively destructor with deletion of allocation
void destroy(pointer p)
{
                alloc.destroy(p);
}

// malloc
pointer allocate(size_type n, const void *hint = 0 )
{
                pointer result = alloc.allocate(n,hint);
                stl_mem_track tmp;
                fprintf(io::getHandle(),"  allocate() addr 0x%p  num:
%d  \n",result, n);
                return result;
};
```

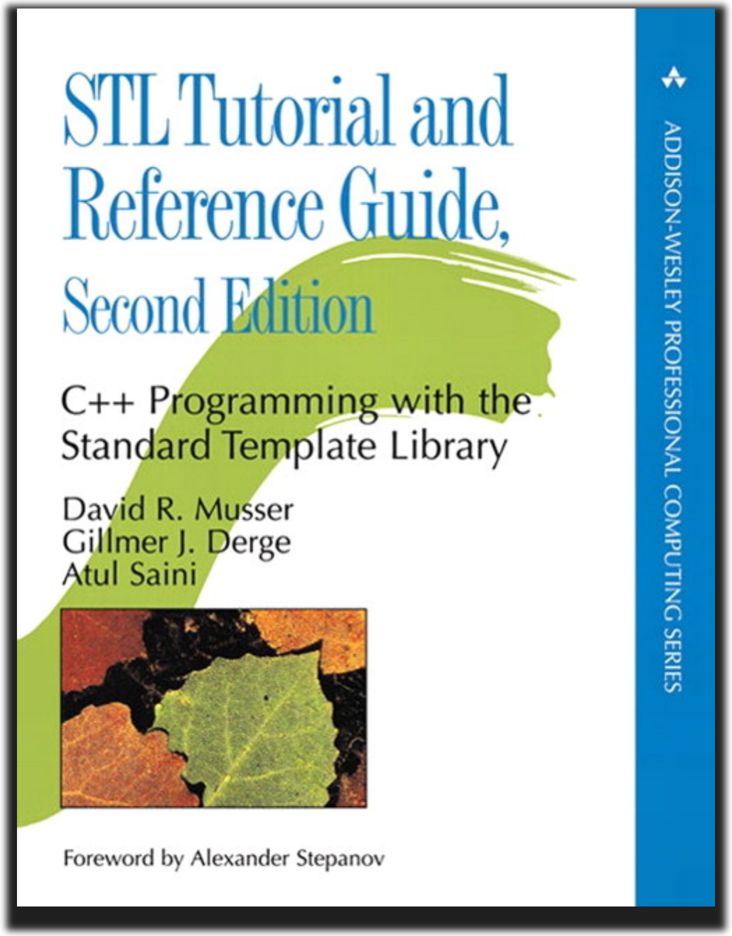
DePaul University

Khan 3/3

```
    // free
    void deallocate(pointer p, size_type n)
    {
        stl_mem_track tmp;
        fprintf(io::getHandle(),"deallocate() addr 0x%p  \n",p );
        alloc.deallocate(p,n);
    };
};

    template < typename T, typename Allocator1, typename U, typename Allocator2 >
    bool operator == ( const logging_allocator<T, Allocator1>& x, const logging_allocator<U, Allocator2>& y)
        {
            return x.alloc == y.alloc;
        };

    template <typename T, typename Allocator1, typename U, typename Allocator2>
    bool operator != ( const logging_allocator<T, Allocator1> &x, const logging_allocator<U, Allocator2> &y)
        {
            return x.alloc != y.alloc;
        };
```

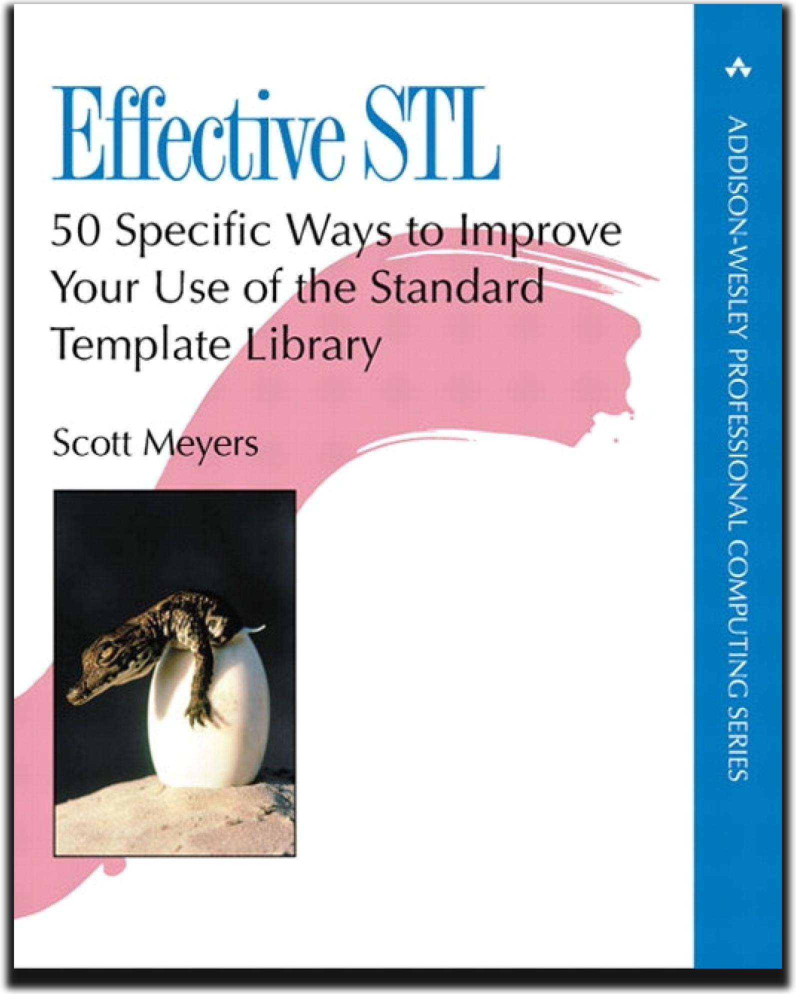
Great book to learn STL

- This is my main reference for STL
  - STL Tutorial and Reference Guide.
- From scratch to overloading
  - Khan’s memory overloading came from this book


STL Tutorial and Reference Guide, Second Edition
C++ Programming with the Standard Template Library
David R. Musser
Gillmer J. Derge
Atul Saini
Foreword by Alexander Stepanov
ADDISON-WESLEY PROFESSIONAL COMPUTING SERIES

Great book to master STL

- Another good reference
  - Effective STL
    - Scott Meyers
      - (He has good hair)
- Many interview questions are STL related
  - You will use STL a lot, master it



Effective STL

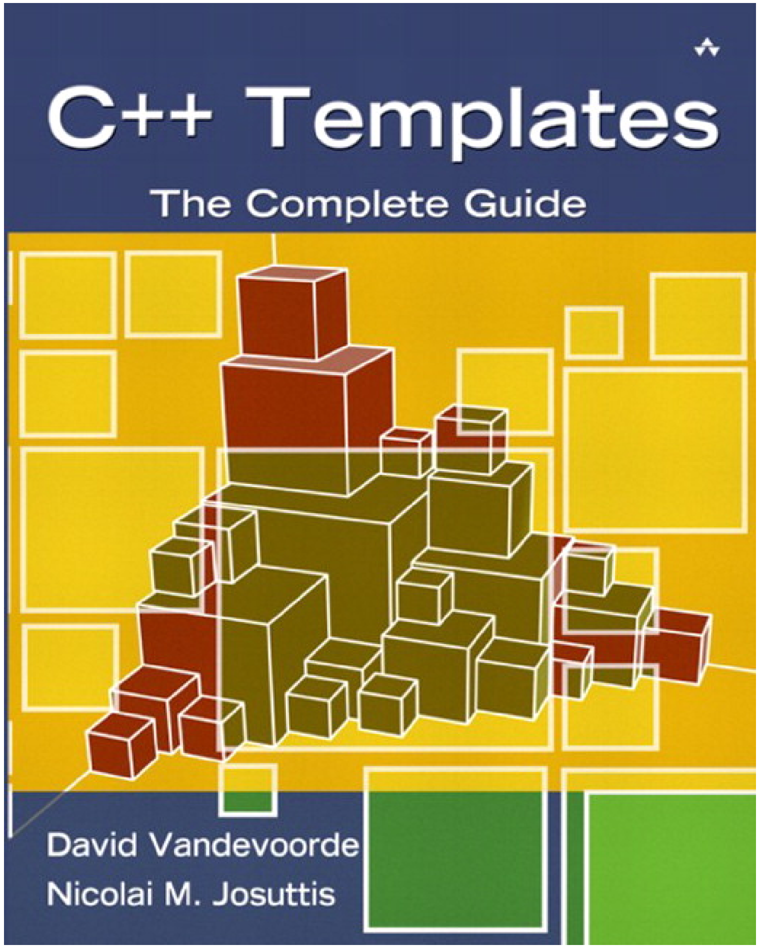
50 Specific Ways to Improve Your Use of the Standard Template Library

Scott Meyers

ADDISON-WESLEY PROFESSIONAL COMPUTING SERIES

The Bible of Templates

- Template book
  - Hard core templates material
  - A lot there to master
  - Take a look



Thank You!

- Questions?

DEPAUL UNIVERSITY

Particle Review

Optimized C++

Ed Keenan

16 May 2018

13.0.5.8.12 Mayan Long Count

DEPAUL UNIVERSITY

Goals

- Particle system review
  - What it does
  - How to Use it
- Explain areas of interest
- Optimization ideas
- Project Deliverables
  - Competition times
  - Logs
- Contest
  - Compare on the same machine.

Who will win?

Keep a log

- Logs are Good!
  - Keep a work log of the tasks your are doing.
  - It’s easy to track your progress
  - Leaves you notes on what you did and how much it helped
  - Leaves bread crumbs for partial credit. ☺

- Logs are Easy to do!
  - Open up a text file or word doc and go.
    - Changelist anyone???
  - Updates take 1-3 minutes
  - Do them after each major refactor
  - Do them at the end of the day

Keep a log

- Changelist: 90863 (Saturday Nov 5) *date stored in changelist automatically*
  - Added const to Vect4D
    - Saved 0.25 ms in release
  - Reworked Matrix to use references instead of pointer
    - Saved 0.05 ms in release
  - Added += operator in Vect4D
    - Reworked code to use this everywhere
    - Save 0.1 ms in release
- Changelist: 90872 (Monday Nov 12)
  - Added SIMD to Vect4D
    - Saved 2.5 ms
  - Cursed Keenan's Name out loud
    - His comments in code were wrong
      - Chased a red herring for 4 hours
      - I'll get him
  - I love Linker errors
    - Sutter come-on give a little more help.

General Ideas…

- Make constant
  - Add ***const*** everywhere
    - Function parameters
    - Constant methods
- Convert pointers to ***references***
  - Gives compiler more options
  - Removes pointer safety checks
- Remove Temporaries
  - Use ***+=, -=, *=, /=*** operators
  - Remember they remove temps

Things to look for…

- Look for invariants
  - Stuff that doesn’t change from loop to loop
    - Remove it
- Look for useless work
  - Remove it
- Dynamic memory timings
  - Is it worth replacing

Containers and Operators

- STL containers
  - Vector
    - Is that the best container
    - Do you need containers
    - Are they being used correctly
- Must overload operators
  - Default constructor
  - Copy Constructor
  - Assignment operator
  - Destructor

Time large sections

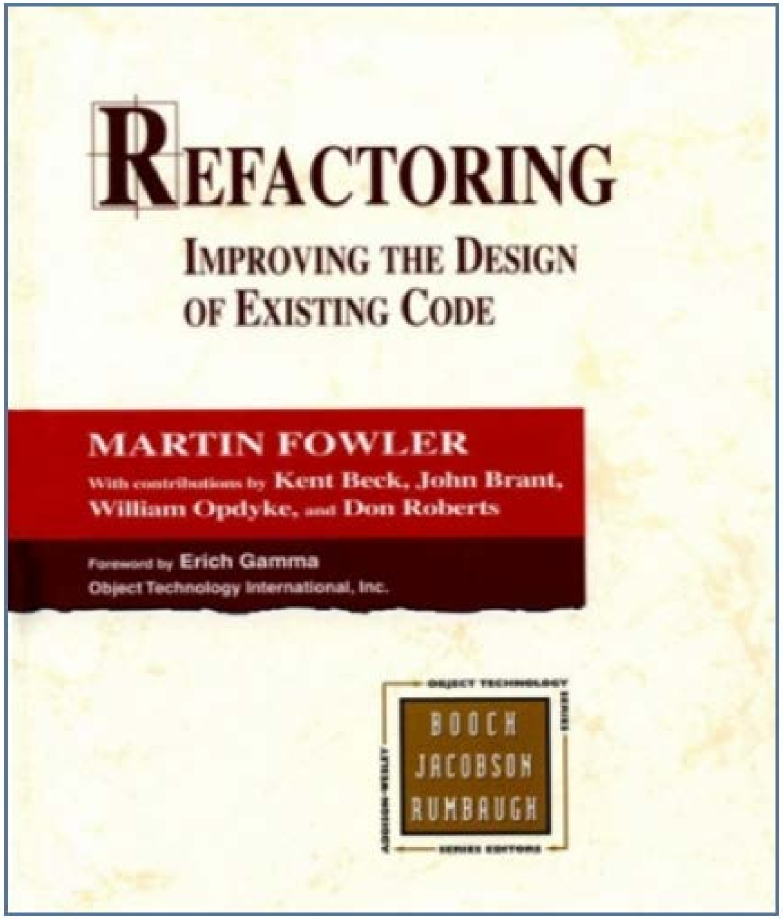
- Time groups and parts of the program
  - Leave Timers in while you develop
- Use #if to enable multiple metrics at a time
  - Control groups of timers with a switch
- Everything changes when you refactor a section.
  - There are side effects.
- Let the data drive you
  - Not your gut feeling or what you read.
  - Timers are KING

Guidelines

- Level Warning ALL
  - Compiles cleanly in Debug and Release
- Timing
  - Need timing metrics for both configurations
  - Need original timing before any modifications
    - Turn off all extra programs on your PC when timing.
- Back up often with notes
  - Use version control
- Logs
  - Turn in Logs every week

Refactoring

- What is it?
- Who has experience with it?
- Do you really understand the principles?



REFACTORING

IMPROVING THE DESIGN OF EXISTING CODE

MARTIN FOWLER

With contributions by Kent Beck, John Brant, William Opdyke, and Don Roberts

Foreword by Erich Gamma
Object Technology International, Inc.

OBJECT TECHNOLOGY SERIES

BOOCH
JACOBSON
RUMBAUGH

SERIES EDITORS

ADDISON-WESLEY

Refactoring

- Definition:
  - *Process of changing a software system in such a way that it does not alter the external behavior of the code yet improves its internal structure*.
- Improving a design after it’s written

What do you Refactor

- For Maintenance reasons
  - Support
  - Development
  - Enhancements
- Architecture Improvements
- Optimizations

Why should you Refactor

- Improves the design of software
- Makes software easier to understand
- Helps you find bugs
- Help you program faster

Rules of the road

- Refactoring changes the programs in small steps.
  - If you make a mistake, its easy to find the bug.
- When you add a feature,
  - Refactor program to make it easier to add the feature.
- Testing is key
  - Make sure the system is understood and well tested, BEFORE you refactor

Important

- Any fool can write code….
  - Good programmers write code that humans understand.

Rule of three

- Three strikes and you refactor
  - Refactor when you add function
  - Refactor when you need to fix a bug
  - Refactor as you do a code review

Bad Smells in Code

- If it stinks, Change it.
  - - Grandma Beck,
    - Discussing child-rearing philosophy

Common Smells

- Duplicated Code
- Long Method
- Large Class
- Long Parameter List
- Divergent Change
- Shotgun Surgery
- Feature Envy
- Data Clumps
- Primitive Obsession
- Switch Statements
- Parallel Inheritance Hierarchies
- Lazy Class
- Speculative Generality

More smells

- Temporary Field
- Message Chains
- Middle Man
- Inappropriate Intimacy
- Alt classes with different interfaces
- Incomplete library class
- Data class
- Refused Bequest
- Comments

Tests

- Yes
- Have some…

Thank You!

- Questions?