

## Re: alloca / \_alloca / dynamic stack memory

---

*Source:* <http://www.tech-archive.net/Archive/VC/microsoft.public.vc.language/2007-03/msg00246.html>

---

- *From:* Michael Crawley <[MichaelCrawley@xxxxxxxxxxxxxxxxxxxxxxxxxxxxxx](mailto:MichaelCrawley@xxxxxxxxxxxxxxxxxxxxxxxxxxxxxx)>
  - *Date:* Thu, 8 Mar 2007 22:24:00 -0800
- 

Lets assume we have a String class with an internal char\* pointer and an unsigned integer for the length. This class has a fixed size. Upon constructing an instance of this type, we pass a variable-sized const char\* that will be copied and set to the internal char\*. So if we used the default new operator, we would be looking at two calls to malloc indirectly...which means the thread had to compete for a shared lock defined within malloc twice. By allocating an instance of String class using new(StackAlloc) operator, we only have to call malloc once when we perform an internal copy of const char\* to be stored in our internal char\* pointer. This might allow for a small perf gain by itself, but if done thousands of times, it can add up. If a heap has been allocated on a per-thread basis, of course use the that heap.

Okay...so what about arrays. No problem... If we go "String\* str = new(StackAlloc) String[N]" we should get this array allocated on the stack dynamically...for each item in the array, the default constructor is called... Everything is fine. To dispose of the array we need something like operator delete[](void\*,StackAlloc), but since C++ does not like parameterized delete operators like the new operator, we need a DisposeArray(void\*,StackAlloc,count) to separate it from the standard delete(void\*,StackAlloc) / Dispose(void\*,StackAlloc) version. DisposeArray would explicitly destruct each item and return. When the function returns the memory is released.

```
String* str = new(StackAlloc) String("msdn");
// Use String
Object::Dispose(str, StackAlloc);

int N = 5;
String* strs = new(StackAlloc) String[N];
// Use Strings
Object::DisposeArray(strs,StackAlloc,N)
```

I know seeing the Dispose/DisposeArray may hurt developers eyes. So to resolve this, we can prepend a management header before every allocation...If we need 23 bytes, we allocate atleast 24 bytes. The one byte header describes whether or not the memory came from the default heap, a specified heap, or a stack. If on a specified heap, prepend a reference to that heap

Re: alloca / \_alloca / dynamic stack memory

at the begining of this memory (23 + 1byte header + specified heap pointer). Since the default heap does not change, working with the stack is the same, and invoking the base class's virtual distructor insures all distructors are called on derived classes, we have all of the ingredience we need to remove the Dispose/DisposeArray methods. When the delete(void\*) or delete[](void\*) operators are called, the compiler will emit the destructors, we process the header, determine the source of memory, and deallocate accordingly.

So...lets say String has a size of 23 bytes (we are not including the dynamic memory needed by its internal char\* pointer)...

```
String* str = new String("msdn"); // (sizeof(String) + 1byte header)
```

```
String* str1 = new(Heap) String("msdn"); // (sizeof(String) + 1byte header + sizeof(void*) ref to heap)
```

```
String* str2 = new(StackAlloc) String("msdn"); // (sizeof(String) + 1byte header)
```

```
delete str; // Compiler emits destructor; Look in header; oh it's on the default heap, free(str)
```

```
delete str1; // Compiler emits destructor; Look in header; oh it's on this heap, heap->Deallocate(str)
```

```
delete str2; // Compiler emits destructor; Look in header; oh it's on the stack, do nothing. deallocation happens when function returns
```

So no more Dispose/DisposeArray mess. Everything looks normal... the developer's eyes have stopped itching. In keeping inline with the new/delete... for every new(StackAlloc) there should also be a delete, only that this is scoped to the current function...

"Doug Harrison [MVP]" wrote:

On Thu, 8 Mar 2007 20:06:13 -0800, Michael Crawley  
<MichaelCrawley@xxxxxxxxxxxxxxxxxxxxxxxxxxxx> wrote:

"Doug Harrison [MVP]" wrote:

On Thu, 8 Mar 2007 12:57:33 -0800, Michael Crawley  
<MichaelCrawley@xxxxxxxxxxxxxxxxxxxxxxxxxxxx> wrote:

```
class _StackAlloc{ };  
const _StackAlloc StackAlloc; // Used only  
to resolve overloaded
```

Re: alloca / \_alloca / dynamic stack memory

method/operator

```
void* Object::operator new(size_t size, const
_StackAlloc& stackAlloc)
{
// Allocation on stack
// ...

// Return Pointer to allocated memory
__asm push pointer onto stack;
__asm jmp (return address); // Jump out of
the current stack frame
};

// Calls destructor, but does not free memory
until the
// current function returns
void Object::Dispose(const StackAlloc&
stackAlloc)
{
// Invoke Distructor
this->~Object(); // Virtual Base destructor

// Perform other operations specific to this
// memory allocation method
// ...
};

class String : public Object
{
public:
// ...
private:
~String(){...}; // Means this type can only be
on allocated on dynamic
memory
};

int main()
{
String str("bad"); // Compiler error, cannot
access private distructor
String* str1 = new(Heap) String("better"); //
Ok
String* str2 = new String("better"); // Ok
String* str3 = new(StackAlloc)
String("better"); // OK, faster than
str1/str2
```

What does String have for data?

Re: alloca / \_alloca / dynamic stack memory

Huh...? Perhaps a char\* pointer and a length field. Perhaps String is not a good example, but I just wanted show how we would like to use the class... A variable-sized array would be more appropriate...

Assume you have your base class, and it does everything you want. Maybe I missed it, but I think it would help to show how String exploits what Object gives it. Also, are there any restrictions on String when used as a class member or the basis of an array type?

--

Doug Harrison  
Visual C++ MVP