



Applied Intelligence  
765 Westwood Drive  
Suite 10A  
Clayton, MO 63105  
(314) 863-4770

# Memory Allocation Methods in Multithreaded Programming in C++ Under Windows NT

**An Applied Intelligence White Paper**

May 19, 2000

Charles R. Calkins

© Applied Intelligence, Inc., 2000

The trademarks and registered trademarks of the corporations mentioned in this publication are the property of their respective holders. Unless otherwise noted, the entire contents of this publication are copyrighted by Applied Intelligence, Inc. and may not be reproduced, stored in a retrieval system, or transmitted in any form or by any means without prior written consent of the publisher.

## **Abstract**

This paper describes methods to allocate memory in C++ in the Win32 environment under Windows NT. Issues relating to memory allocation from multiple threads are highlighted.

## **Keywords**

- memory allocation
- C++
- multithread
- Win32
- Windows NT

## Table of Contents

Abstract .....	i
Keywords.....	i
Executive Summary .....	1
Introduction .....	1
Win32 API under Windows NT – The Allocation Methods .....	1
The Methods Suitable for “Small” Allocations .....	1
<i>new/delete</i> .....	1
<i>malloc()/free()</i> .....	2
<i>GlobalAlloc()/GlobalFree()</i> .....	2
<i>HeapAlloc()/HeapFree()</i> .....	2
The Methods Not Suitable for “Small” Allocations .....	3
<i>VirtualAlloc()/VirtualFree()</i> .....	3
<i>CreateFileMapping()/CloseHandle()</i> .....	3
The Benchmark.....	3
The Benchmark Application.....	3
The Benchmark Results .....	4
<i>new/delete</i> and <i>malloc()/free()</i> .....	4
<i>GlobalAlloc()/GlobalFree()</i> and <i>HeapAlloc()/HeapFree()</i> Using the Process Heap.....	5
<i>HeapAlloc()/HeapFree()</i> Using a Preallocated Thread Local Heap.....	6
<i>HeapAlloc()/HeapFree()</i> Using a Preallocated Thread Local Heap Without Synchronization..	7
Listing 1 .....	8
Summary .....	10

## Executive Summary

Under Windows NT a number of memory allocation mechanisms are provided. The most obvious for use in C++ programming isn't the best, however, for optimal performance in multithreaded applications.

## Introduction

The Win32 environment provides a number of ways for an application to allocate memory. Not all of these methods perform well when multiple threads are used, however, and this white paper describes several of the memory allocation methods and provides a performance comparison between them.

## Win32 API under Windows NT – The Allocation Methods

A C++ program written for the Win32 API has a number of methods available to it for the allocation of memory ranging from C++ language features, to the C library, to the Win32 API itself. The methods focused on in this white paper are suitable for the allocation of "small" blocks of memory as is typical in C++ programs – objects which are frequently used such as strings, linked list elements, and the like are closer to tens of bytes rather than millions of bytes long. Other memory allocation mechanisms exist which will be touched upon, but not described in detail as their usage precludes "small" block direct allocation, although the methods could form the allocation core of a custom suballocation scheme.

## The Methods Suitable for "Small" Allocations

### *new/delete*

The standard method for memory allocation in C++ is by using the *new* and *delete* language keywords. *new* not only allocates storage for an object or array of objects, but also invokes the object's constructor to initialize the object to a programmer-defined state. *delete* performs an analogous but opposite operation – it executes the destructor of the object to perform any needed cleanup, then frees the memory that had been allocated. Out of the various memory allocation methods, only *new* and *delete* call an object's constructor and destructor, respectively, as *new* and *delete* are the only memory allocation mechanism inherent in C++.

Allocation via *new* is type safe – the pointer returned by *new* is to the same type as the object as being constructed.

```
MyClass *c=new MyClass;
```

allocate and initialize one object of type MyClass

```
int *i=new int[10];
```

allocate an array of 10 objects (integers)

```
delete c;
```

call the MyClass destructor and free the memory

```
delete [] i;
```

delete the array of objects by calling the destructor of each if a destructor exists, then freeing the memory

## *malloc()/free()*

The C library's standard functions to allocate and free memory are through *malloc()* and *free()*. *malloc()* allocates a block of memory of a specified size (specified in bytes), and *free()* frees the allocated block. The pointer returned from *malloc()* is of type `void *`, so an explicit cast must be made to the desired type.

```
int *i=(int *)malloc(10*sizeof(int));           allocate memory for 10 integers
free(i);                                       free the memory
```

## *GlobalAlloc()/GlobalFree()*

The Windows API provides multiple methods for allocating memory, the oldest by *GlobalAlloc()* which existed even prior to the Win32 API. *GlobalAlloc()* is frequently used to return a handle to a reserved region of memory. As Windows is free to move the memory block as it sees fit, to use the block it must be pinned to a specific address by *GlobalLock()* - *GlobalLock()* returns a `void *`. When the memory has been used, *GlobalUnlock()* unpins the block, and *GlobalFree()* frees it. The *GlobalLock()/GlobalUnlock()* step can be bypassed by providing the parameter `GMEM_FIXED` to *GlobalAlloc()* - the returned value is then a pointer instead of a handle which is already pinned. *GlobalFree()* is still used to free the allocated memory.

Prior to the Win32 API the functions *LocalAlloc()* and *LocalFree()* could be used to allocate and free memory from a local heap, but in the Win32 API the global and local functions access the same heap.

As per the April 2000 Microsoft Developer Network Library, new applications should use the heap functions, rather than *GlobalAlloc()* for memory allocation.

```
int *i=(int *)GlobalAlloc(GMEM_FIXED, 10*sizeof(int));
GlobalFree(i);
```

## *HeapAlloc()/HeapFree()*

*HeapAlloc()* allows memory to be allocated from a specified memory heap. This heap is either global for the entire process - obtained by calling *GetProcessHeap()* - or from a custom heap that has been allocated by *HeapCreate()*. Memory from a heap is freed by specifying the originating heap and memory pointer to *HeapFree()*.

Allocating memory via the heap functions offer two specific benefits. The first is that the heap size can be specified. A minimum can be given, as well as a specified maximum or 0 implying the heap can grow as needed. The second benefit is that by specifying `HEAP_NO_SERIALIZE` to the heap functions, the heap can be used in a way that accesses to it are not synchronized. If it is certain that only a single thread of execution will be using the heap, synchronization to the heap can be disabled, improving performance.

```
int flag=0;                                     created heaps have synchronization
int flag= HEAP_NO_SERIALIZE;                   created heaps do not have
                                              synchronization

HANDLE heap=GetProcessHeap();                 use the process heap
HANDLE heap=HeapCreate(flag, 4096, 1048576);   create a heap with minimum size of
                                              4096 bytes and maximum size of 1MB
HANDLE heap=HeapCreate(flag, 4096, 0);        create a heap with a minimum size
                                              of 4096 bytes and no maximum size
```

<code>int *i=(int *)HeapAlloc(heap, flag, 10*sizeof(int));</code>	allocate from the heap
<code>HeapFree(heap, flag, i);</code>	free the memory
<code>HeapDestroy(heap);</code>	free the heap (don't call this if the heap was obtained from <code>GetProcessHeap()</code> )

## **The Methods Not Suitable for “Small” Allocations**

### *VirtualAlloc()/VirtualFree()*

For greater control over the allocation of memory, *VirtualAlloc()* and *VirtualFree()* are provided. Memory, when allocated, may be initially committed, or may be reserved but not committed until a later point in time. *VirtualAlloc()* can also provide access to a specific memory region, or memory outside of the 32 bit address space.

*VirtualAlloc()* does have restrictions on the alignment of the starting address, and allocates memory in terms of pages, thus precluding its direct use for the allocation of “small” objects.

### *CreateFileMapping()/CloseHandle()*

Win32 allows mapping a physical file on disk, or a segment of the paging file, as a memory region. *CreateFileMapping()* is used to identify the file or paging file segment to be used for the mapping, and a handle returned. This handle is pinned in memory through the use of *MapViewOfFile()* which provides a void \* pointer to the memory, a function analogous to *GlobalLock()*. When the memory is to be discarded, *UnmapViewOfFile()* unpins the memory block, and calling *CloseHandle()* on the handle returned by *CreateFileMapping()* terminates the mapping.

Generally one would not map many small files(or paging file regions) but instead would map a single one and suballocate from it, thus precluding its use for direct “small” allocations.

## **The Benchmark**

With so many memory allocation methods available it is important to know the performance of each, so applications, in particular multithreaded ones, can execute efficiently. Contrasting the execution time of various methods, relative to allocation size and number of concurrent threads, demonstrates which methods should be used in which situations.

### **The Benchmark Application**

In order to identify which methods should be used over others in a multithreaded application, a test program was developed (Listing 1). This program clocks the time spent allocating and freeing memory blocks of a specific size using the various methods described above for “small” block allocation. The user performing the benchmark, via command line switches, can control the number of allocations and frees performed, the size of the allocated block, which method is used for the allocation and free, and how many concurrent threads will be performing the allocation and frees.

This program was run to clock the time spent on 2<sup>20</sup> repeated allocations and frees of block sizes ranging from 1 byte to 64 kilobytes. The results were repeatedly gathered from 1, 2, 3

and 4 concurrent threads to measure the impact of multithreaded access on memory allocation. The results were obtained from runs on a dual 450MHz Pentium II with 256MB of RAM running Windows NT 4 SP6a. The machine was idle other than for standard NT services, and approximately 160MB of RAM was free when the runs were made, ensuring no paging was necessary. The compiler used was Visual C++ 6.0 SP3 and the application compiled for maximum speed in Release mode.

## The Benchmark Results

### *new/delete* and *malloc()/free()*

Visual C++ 6.0 implements a suballocation strategy for memory allocations. A small block allocator is used for allocations of size 984<sup>1</sup> bytes and lower where allocations are serviced by blocks from a previously allocated structure, while larger requests call *HeapAlloc()* directly, where *HeapAlloc()* allocates from a custom heap created for the C runtime. Both *new/delete* and *malloc()/free()* are implemented using this allocation scheme, so timings for different allocation sizes are virtually identical. The timings for *new/delete* are shown here. One two four threads are presented, and the block allocation size and time (in microseconds) to perform that allocation.

One Thread		Two Threads		Three Threads		Four Threads	
Bytes	Time (µs)	Bytes	Time (µs)	Bytes	Time (µs)	Bytes	Time (µs)
1	1.192	1	22.828	1	30.935	1	23.932
2	1.192	2	22.918	2	22.918	2	23.633
4	1.178	4	23.305	4	30.741	4	23.603
8	1.178	8	22.828	8	30.771	8	23.811
16	1.177	16	22.978	16	31.352	16	24.051
32	1.192	32	23.306	32	31.038	32	23.902
64	1.192	64	23.365	64	31.487	64	23.931
128	1.192	128	23.275	128	30.950	128	24.156
256	1.192	256	22.992	256	31.054	256	24.303
512	1.206	512	22.710	512	31.054	512	23.976
1024	1.342	1024	4.858	1024	10.416	1024	10.639
2048	1.356	2048	4.575	2048	7.674	2048	8.910
4096	1.356	4096	4.575	4096	8.196	4096	9.030
8192	1.356	8192	4.589	8192	8.018	8192	8.240
16384	1.370	16384	4.694	16384	7.465	16384	6.914
32768	1.446	32768	42.379	32768	52.854	32768	63.763
65536	46.580	65536	106.677	65536	105.336	65536	104.219

In the single thread case performance through 32KB allocations performance is about the same. At 64KB, *HeapAlloc()* becomes slow. For more than a single thread the small block allocator also performs badly due to the attempted simultaneous access by multiple threads, and strangely three threads are worse than two or four. Performance improves after the 984

<sup>1</sup> The user's request is first padded by 36 bytes, then that value is compared to 1016 - if less than or equal, the small block allocator is used. 1016 is 1024 minus 8 bytes.

byte boundary is passed as the small block allocator is no longer used and *HeapAlloc()* is called directly, but in the multithreaded case *HeapAlloc()* begins to perform badly at 32KB.

Note that the time listed per allocation is actually over all threads – for instance, if a single thread takes a total of N time units to perform a task, ideally two threads could perform the task, if perfectly parallelizable, in N/2 time units. The purpose of the sample program is to allocate and free a given block size  $2^{20}$  times – two threads should perform that task twice as fast as a single thread should. More than two threads will likely not improve performance as the test machine only has two CPUs.

This test run however shows that multiple threads do not improve performance, they actually degrade it due to the heap contention. In the instances where the suballocator are used, performance is as much as 30 times worse with multiple threads than with just one.

### *GlobalAlloc()/GlobalFree()* and *HeapAlloc()/HeapFree()* Using the Process Heap

As Win32 implements a single heap by default for a process, the results for *GlobalAlloc()/GlobalFree()* and *HeapAlloc()/HeapFree()* when the heap obtained by calling *GetProcessHeap()* are used are virtually the same. The results for *HeapAlloc()/HeapFree()* are shown here.

One Thread		Two Threads		Three Threads		Four Threads	
Bytes	Time (µs)	Bytes	Time (µs)	Bytes	Time (µs)	Bytes	Time (µs)
1	0.611	1	1.177	1	1.282	1	1.222
2	0.596	2	1.192	2	1.192	2	1.222
4	0.596	4	1.237	4	1.251	4	1.252
8	0.596	8	1.178	8	1.192	8	1.237
16	0.610	16	1.386	16	1.282	16	1.281
32	0.611	32	1.222	32	1.206	32	1.223
64	0.611	64	1.222	64	1.237	64	1.237
128	0.611	128	1.236	128	1.237	128	1.237
256	0.611	256	1.206	256	1.223	256	1.236
512	0.610	512	1.237	512	1.223	512	1.222
1024	0.909	1024	3.919	1024	4.441	1024	4.932
2048	0.910	2048	3.903	2048	3.889	2048	4.158
4096	0.910	4096	3.948	4096	4.113	4096	4.663
8192	0.909	8192	3.979	8192	4.068	8192	4.485
16384	0.910	16384	4.008	16384	4.098	16384	4.560
32768	0.909	32768	83.700	32768	60.201	32768	87.842
65536	46.030	65536	117.376	65536	105.187	65536	107.019

Overall, performance is very good even in the multithreaded case. Performance only drops when multiple threads are used at 32KB and higher allocation sizes. Comparing these results to the previous set shows the small block allocator really is at fault – bypassing it completely improves performance significantly.

Performance is still worse with multiple threads however. For up to 512 byte blocks two threads take twice, not half, as long as one thread. For 1K to 16K blocks, two threads take four times as long, and for 32K 83 times as long. Unlike the previous case, however, more than two threads do not additionally degrade performance.

## HeapAlloc()/HeapFree() Using a Preallocated Thread Local Heap

If pointers to data need not be shared between threads, each thread can allocate its own local heap using *HeapCreate()* with the performance improving as compared to the global heap. As the heaps are local to each thread there is no contention.

One Thread		Two Threads		Three Threads		Four Threads	
Bytes	Time (µs)	Bytes	Time (µs)	Bytes	Time (µs)	Bytes	Time (µs)
1	0.611	1	0.313	1	0.298	1	0.314
2	0.596	2	0.299	2	0.313	2	0.299
4	0.610	4	0.299	4	0.313	4	0.299
8	0.611	8	0.299	8	0.313	8	0.298
16	0.610	16	0.314	16	0.313	16	0.313
32	0.611	32	0.313	32	0.313	32	0.313
64	0.611	64	0.313	64	0.313	64	0.313
128	0.610	128	0.313	128	0.298	128	0.314
256	0.611	256	0.299	256	0.299	256	0.313
512	0.610	512	0.299	512	0.313	512	0.313
1024	0.983	1024	0.492	1024	0.491	1024	0.492
2048	0.939	2048	0.477	2048	0.477	2048	0.462
4096	1.058	4096	0.537	4096	0.536	4096	0.537
8192	0.954	8192	0.477	8192	0.477	8192	0.491
16384	0.939	16384	0.477	16384	0.477	16384	0.492
32768	0.938	32768	0.477	32768	0.477	32768	0.492
65536	45.985	65536	96.798	65536	82.120	65536	86.651

Performance actually seems to be better in the multithreaded cases until 64KB, then performance drops again when multiple threads are used, even though the heaps are local to each thread. The 64KB case with a local heap is still faster than when using a global heap though.

Timings for this run are more along the lines of what one would expect in the multithreaded cases. For two threads, allocation time is about half of the time elapsed for one thread to complete the task. More than two threads do not help or hinder the performance for block sizes less than 64K as the test machine contains only two CPUs. The 64K block size is still a problem however in the multithreaded case - additional threads increase execution time, rather than decreasing it.

## HeapAlloc()/HeapFree() Using a Preallocated Thread Local Heap Without Synchronization

As each heap is local to each thread, synchronization on heap access can be removed. Since only one thread will be accessing a given heap at a given time, there is no need to guarantee mutually exclusive access.

One Thread		Two Threads		Three Threads		Four Threads	
Bytes	Time (μs)	Bytes	Time (μs)	Bytes	Time (μs)	Bytes	Time (μs)
1	0.582	1	0.298	1	0.283	1	0.299
2	0.581	2	0.283	2	0.299	2	0.283
4	0.581	4	0.283	4	0.299	4	0.298
8	0.581	8	0.283	8	0.298	8	0.299
16	0.566	16	0.298	16	0.283	16	0.298
32	0.566	32	0.299	32	0.283	32	0.298
64	0.566	64	0.299	64	0.283	64	0.298
128	0.566	128	0.283	128	0.299	128	0.283
256	0.566	256	0.283	256	0.283	256	0.283
512	0.566	512	0.283	512	0.283	512	0.283
1024	0.522	1024	0.268	1024	0.254	1024	0.254
2048	0.566	2048	0.283	2048	0.282	2048	0.283
4096	0.656	4096	0.328	4096	0.327	4096	0.328
8192	0.522	8192	0.268	8192	0.268	8192	0.269
16384	0.506	16384	0.268	16384	0.269	16384	0.268
32768	0.522	32768	0.253	32768	0.268	32768	0.269
65536	45.865	65536	95.576	65536	81.271	65536	86.411

As in the previous case, timings are more in line with what is expected from multiple threads. Overall performance for 32KB and smaller block sizes is even better, as no time is spent checking a synchronization primitive. 64KB blocks are still a problem however.

## Listing 1:

```
#include <windows.h>
#include <fstream.h>
#include <string.h>
#include <time.h>

long nalloc=1024;
long nthreads=1;
long size=64;
long type=0;

#define A0001(x) { x }
#define A0002(x) A0001(x) A0001(x)
#define A0004(x) A0002(x) A0002(x)
#define A0008(x) A0004(x) A0004(x)
#define A0016(x) A0008(x) A0008(x)
#define A0032(x) A0016(x) A0016(x)
#define A0064(x) A0032(x) A0032(x)
#define A0128(x) A0064(x) A0064(x)
#define A0256(x) A0128(x) A0128(x)
#define A0512(x) A0256(x) A0256(x)
#define A1024(x) A0512(x) A0512(x)

#define SMALLHEAP 4096
#define BIGHEAP 16*1024*1024

void WINAPI allocthread(unsigned long *) {
    switch (type) {
        case 0: // new/delete
            while (InterlockedDecrement(&nalloc)>0) {
                A1024(char *x=new char[size]; delete [] x);
            }
            return;
        case 1: // malloc/free
            while (InterlockedDecrement(&nalloc)>0) {
                A1024(char *x=(char *)malloc(size*sizeof(char)); free(x));
            }
            return;
        case 2: // GlobalAlloc/GlobalFree
            while (InterlockedDecrement(&nalloc)>0) {
                A1024(char *x=(char *)GlobalAlloc(GMEM_FIXED, size*sizeof(char)); GlobalFree(x));
            }
            return;
        case 3: // HeapAlloc/HeapFree using the process heap
            {
                HANDLE heap=GetProcessHeap();
                while (InterlockedDecrement(&nalloc)>0) {
                    A1024(char *x=(char *)HeapAlloc(heap, 0, size*sizeof(char)); HeapFree(heap,0,x));
                }
            }
            return;
        case 4: // HeapAlloc/HeapFree using a created heap of minimal size
            {
                HANDLE heap=HeapCreate(0, SMALLHEAP, 0);
                while (InterlockedDecrement(&nalloc)>0) {
                    A1024(char *x=(char *)HeapAlloc(heap, 0, size*sizeof(char)); HeapFree(heap,0,x));
                }
                HeapDestroy(heap);
            }
            return;
        case 5: // HeapAlloc/HeapFree using a created heap of large size
            {
                HANDLE heap=HeapCreate(0, BIGHEAP, 0);
                while (InterlockedDecrement(&nalloc)>0) {
                    A1024(char *x=(char *)HeapAlloc(heap, 0, size*sizeof(char)); HeapFree(heap,0,x));
                }
                HeapDestroy(heap);
            }
            return;
    }
}
```

```

case 6: // HeapAlloc/HeapFree using a created heap of minimal size, no synchronization
{
    HANDLE heap=HeapCreate(HEAP_NO_SERIALIZE, SMALLHEAP, 0);
    while (InterlockedDecrement(&nalloc)>0) {
        A1024(char *x=(char *)HeapAlloc(heap, HEAP_NO_SERIALIZE, size*sizeof(char));
        HeapFree(heap,HEAP_NO_SERIALIZE,x));
    }
    HeapDestroy(heap);
}
return;
case 7: // HeapAlloc/HeapFree using a created heap of large size, no synchronization
{
    HANDLE heap=HeapCreate(HEAP_NO_SERIALIZE, BIGHEAP, 0);
    while (InterlockedDecrement(&nalloc)>0) {
        A1024(char *x=(char *)HeapAlloc(heap, HEAP_NO_SERIALIZE, size*sizeof(char));
        HeapFree(heap,HEAP_NO_SERIALIZE,x));
    }
    HeapDestroy(heap);
}
return;
default:
;
}
}

void main(int argc, char *argv[]) {
    int i=1;
    while (i<argc) {
        if (!strcmp(argv[i], "-nalloc")) {
            i++;
            nalloc=atol(argv[i++]);
        } else
        if (!strcmp(argv[i], "-nthreads")) {
            i++;
            nthreads=atol(argv[i++]);
        } else
        if (!strcmp(argv[i], "-size")) {
            i++;
            size=atol(argv[i++]);
        } else
        if (!strcmp(argv[i], "-type")) {
            i++;
            type=atol(argv[i++]);
        } else
            i++;
    }

    cout << "Allocation and frees of " << 1024*nalloc << " blocks of " << size << " bytes using "
        << nthreads << " thread" << (nthreads>1?"s":"" ) << endl;
    cout << "using ";
    switch (type) {
    case 0: cout << "new/delete"; break;
    case 1: cout << "malloc/free"; break;
    case 2: cout << "GlobalAlloc/GlobalFree"; break;
    case 3: cout << "HeapAlloc/HeapFree using the process heap"; break;
    case 4: cout << "HeapAlloc/HeapFree using a created heap of " << SMALLHEAP << " bytes";
        break;
    case 5: cout << "HeapAlloc/HeapFree using a created heap of " << BIGHEAP << " bytes"; break;
    case 6: cout << "HeapAlloc/HeapFree using a created heap of " << SMALLHEAP <<
        " bytes without synchronization"; break;
    case 7: cout << "HeapAlloc/HeapFree using a created heap of " << BIGHEAP <<
        " bytes without synchronization"; break;
    case 8: cout << "VirtualAlloc/VirtualFree with commit"; break;
    default:
        ;
    }
    cout << endl;
}

```

```
HANDLE *threads=new HANDLE[nthreads];

for (int i=0; i<nthreads; i++)
    threads[i]=CreateThread(0,0, (LPTHREAD_START_ROUTINE)allocthread, 0,CREATE_SUSPENDED ,0);

clock_t start=clock();
for (i=0; i<nthreads; i++)
    ResumeThread(threads[i]);
WaitForMultipleObjects(nthreads, threads, TRUE, INFINITE);
double elapsed=(clock()-start)/(double)CLOCKS_PER_SEC;

cout << "Elapsed: " << elapsed << endl;

for (i=0; i<nthreads; i++)
    CloseHandle(threads[i]);

delete [] threads;
}
```

## Summary

This white paper has shown that, for optimal performance in multithreaded applications, the obvious memory allocation method isn't always the best.