THE SUBALLOCATOR ALGORITHMS

Francisco José Tapia (fjtapia@gmail.com)

1. - ALLOCATORS. DESCRIPTION AND PROBLEMS
    1.1.- PROBLEM 1 THE SPEED
    1.2.- PROBLEM 2 THE MEMORY
    1.3.- PROBLEM 3 THE CACHE PERFORMANCE

2.- SUBALLOCATOR. FUNCTIONAL DESCRIPTION OBJETIVES

3.- SUBALLOCATOR. INTERNAL DESCRIPTION AND ALGORITHMS
    3.1.- THE POOL ALLOCATOR
    3.2.- THE HEAP
    3.3.-OTHERS ALGORITHMS

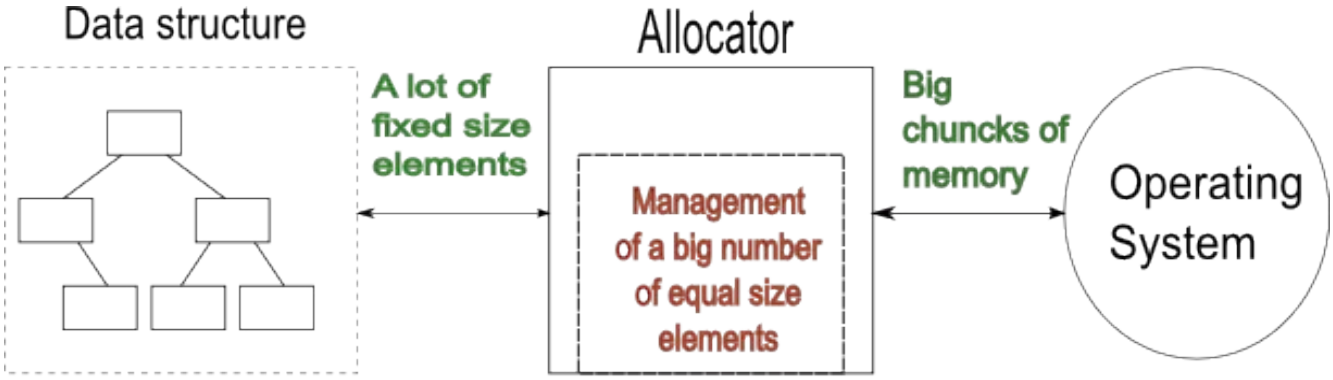
1.- ALLOCATORS. DESCRIPTION AND PROBLEMS

The allocator is the data structure defined in the STL, which is the interface between the data structures and the memory provided generally by the Operating System.

The allocators have a well defined interface. The data structures don't need to know anything about the allocator except the interface. The allocator manages the memory received from the Operating System, and the memory requested in the allocate operations and the memory returned in the deallocate operations.

The time spent and the memory used are the main measures of the quality of an allocator. To write an allocator for to manage a wide range of size elements and a big number of elements of each size in a fast way, it is a very difficult task.

Some years ago, only supercomputers have 4 GB of memory. Now, the cheap laptop have 6 GB, and is expected the memory of the computers grow. Many allocators of modern compilers are not well prepared for to manage hundred of millions of elements. With a small number of elements they are efficient, but when have a great number of elements, the speed down, and appear problems with the memory management. It is a challenge for the HW designers and mainly for the SW designers.



A Pool Allocator, is an allocator which request to the Operating System more memory than the needed. This allocator don't need request memory to the operating system for each allocation. This improve the speed of the allocator, but many of them increment the memory consumption of the program because have memory owned but don't used.

1.1.- PROBLEM 1. THE SPEED

A very hard test for the allocators are the data structures which need a very high number of elements of the same size, like the STL data structures list, set, multiset, map or multimap.

The solution proposed for this problem are the custom allocators. These allocators are very fast, but many of them present additional problems related with the memory consumption.

1.2.- PROBLEM 2. THE MEMORY

The allocators request memory to the operating system when receives petitions of allocation , but many of them, when the memory is deallocate don't return memory to the operating system, and that memory is owned by the allocator but not used.

If you have a small number of elements, you have a small problem, small resources and small time operations. But, if you have several millions of elements allocated, perhaps you are using several GB of memory. Running a program with GB of memory don't used, because the allocator don't return the memory request, is a great waste of resources. Unacceptable if you are running the program continuously many hours, or if you don't have many resources in your machine, like a mobile phone or a tablet.

The problem for the allocators is how to know when the chunk of memory requested to the operating system is empty and don't have any element allocated in that chuck. If the chuck have capacity for to allocate many elements of equal size, to check if the chuck is empty can be a hard work. Only when the chuck is empty the allocator can return it to the operating system, in order to reduce the memory consumption of the program.

If you use a custom allocator and that allocator don't return well the memory, you have an allocator for other data structures like vectors, and the custom allocator. Each allocator have its own memory and don't share between them. This produce a high consumption of memory because the memory don't use can't be shared between them.

Other additional problem is the information around the memory allocate. Many allocators, when you request memory for to allocate a double number. The allocator return you a pointer to a 8 bytes of memory, but around that memory you have additional information used by the allocator, for to deallocate that memory. By example, allocate 50.000.000 elements of 64 bits, the std::allocator of GCC 4.6 use 1.56 Gigas of memory and the boost::fast_pool_allocator 0.52 Gigas

1.3.- PROBLEM 3 THE CACHE PERFORMANCE

The last problem associated to the allocators is the cache performance. When you have a big data structure (imagine a std::set with 50.000.000 elements), travel the structure for to find an element or for to insert an element, you must cross though many nodes. Calculate the hit rate of the cache is extremely difficult. But some measures can provide us a very useful information.

Take in mind, that the performance can have great variations, depending of the processor, and mainly of the cache size.

On my computer in the allocation of 50.000.000 elements of 64 bits, on Linux 64 bits

NAME	**Time Spent**	**Time of individual allocation**
std::allocator	1.44 seconds	28.8 nanoseconds.
std::allocator+ suballocator	0.76 seconds	15.2 nanoseconds

If we check the time with the std::set and different allocators in the insertion of 30.000.000 random elements of 64 bits (In the two test insert the same sequence of numbers)

NAME	Time Spent	Time of individual insertion
std::allocator	69.78 seconds	2326 nanoseconds.
std::allocator + suballocator	39.65 seconds	1321 nanoseconds

The time difference between the allocation and insertion of one element in a std::set with the two allocators is 1005 nanoseconds. The time difference between the two allocators is 13.6 nanoseconds.

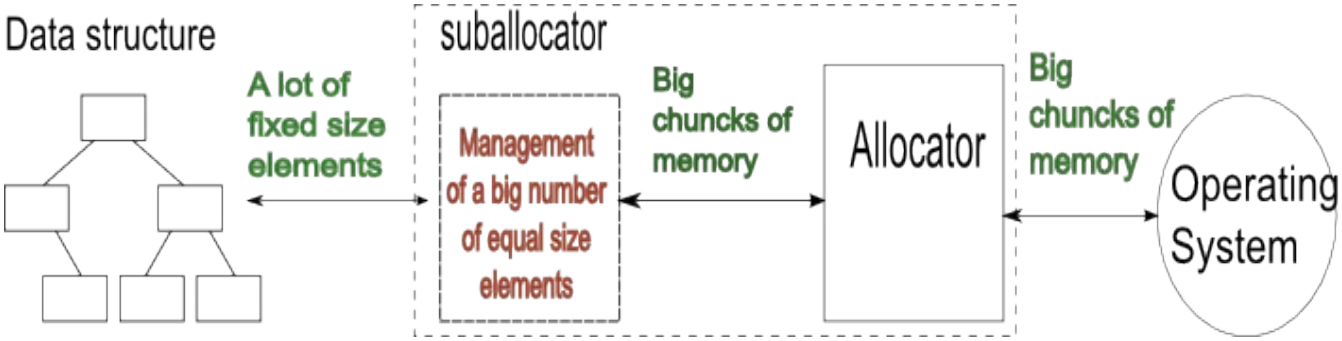
1005 - 13.6 = 991.4 nanoseconds

What is the reason of this difference of 991.4 nanoseconds ? The response is the cache performance due to the data locality improvement.

2.- SUBALLOCATOR . FUNCTIONAL DESCRIPTION OBJETIVES

The suballocator born with the idea of tray to resolve, or at least mitigate, the problems associated with the allocators.

The suballocator is a layer over the allocator. It is a mechanism for to manage in a fast way a greater number (hundred millions) of elements with the same size. The suballocator always return the first element free. This improve the cache performance and the speed of the data structures.



The suballocator receives an allocator as template parameter. When the suballocator needs memory, request memory from the allocator, and when the memory is not used, is returned to the allocator. This simple memory schema permit to the allocators return memory to the Operating System and decrease the memory used by the program, as demonstrate the programs in the benchmarks, and you can see in the benchmark point.

The suballocator present the same interface than the STL allocator. Form the view point of the data structures, the suballocator is other allocator.

With the suballocator

A) We have a very **fast allocation** *(several times faster than GCC 4.6 std::allocator and Visual Studio 10 std::allocator *See details in the* Suballocator Benchmark)*

B) **Return memory to the allocator**, for to be used by others types of data. Many allocators, return this memory to the Operating System, and decrease the memory used by the program

C) You **can use with any allocator** if according with the STL definition. The suballocator provides speed and memory management to any allocator.

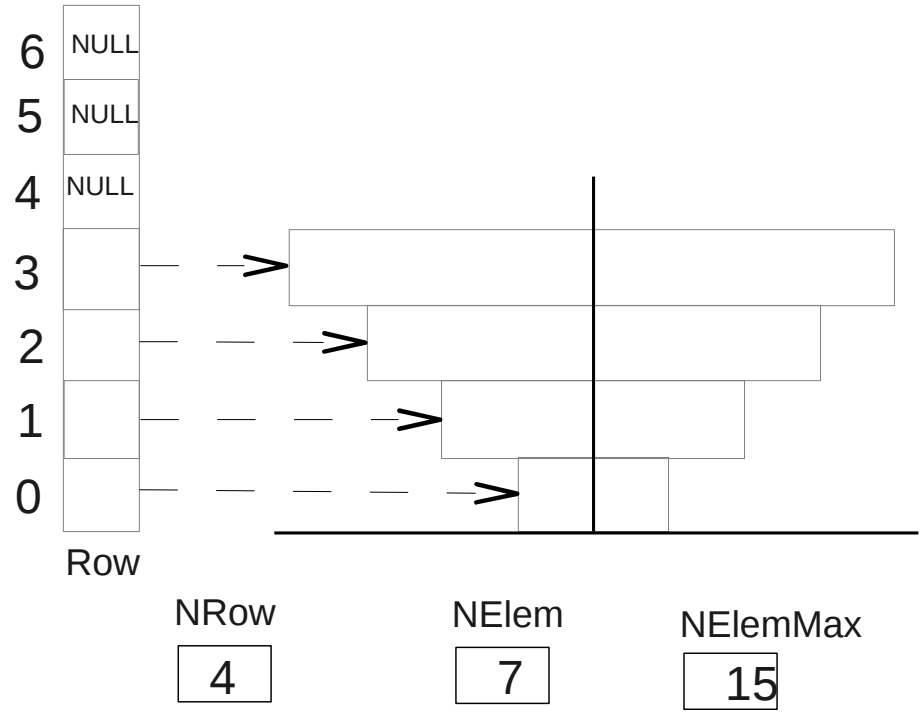
D) The suballocator always provide the first position free. With this we obtain a very compact data areas, which **improve the cache performance** **(***30% saving time in the insertion in a std::set. See details in the Suballocator Benchmarks)*

The secret is the algorithm.

3.- INTERNAL DESCRIPTION AND ALGORITHMS

3.1.- THE POOL ALLOCATOR

The pool allocator is like a Hanoi tower but upside down.



This pool is static and used by all the suballocators and data structures with elements of the same size. If you have a std::list and a std::map and the two need elements of the same size, they share the pool. You have a static pool for each size element requested by the data structures.

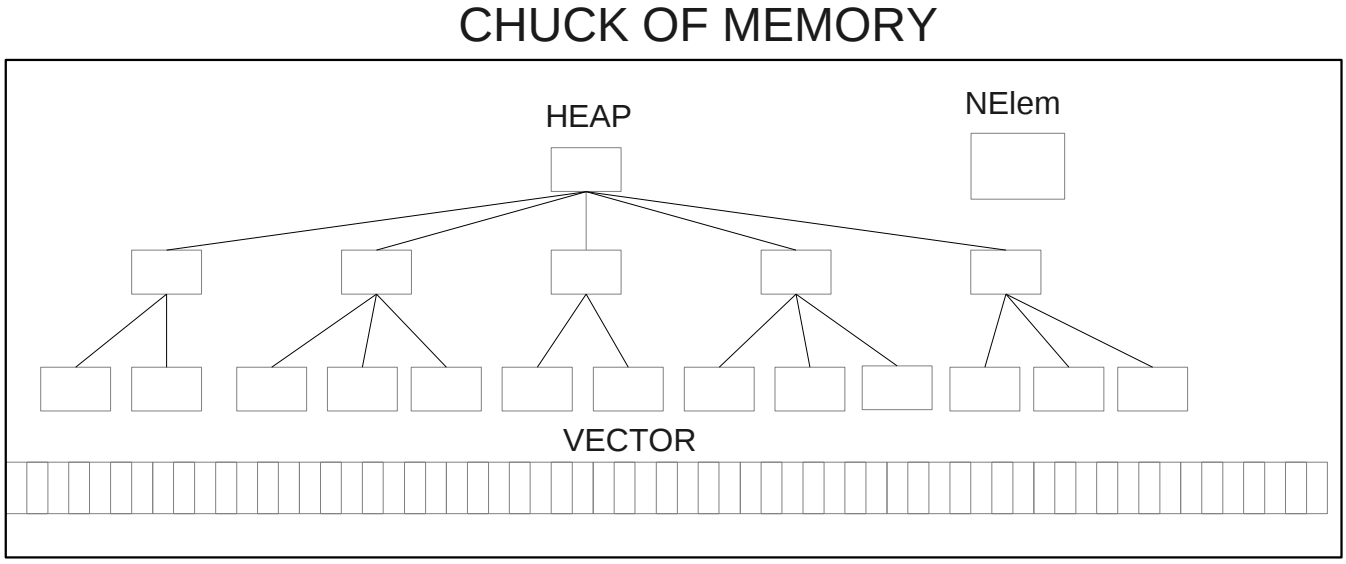
It is static and common, because must be possible to do the splice and merge operations of the std::list, and for move elements from one data structure to other as need with the rvalues. This permit to the iterators to the elements remain valid after these operations. They pointed to the same element, but in different data structure.

When the suballocator need memory, request to the allocator a chuck of memory for to allocate 1 element (2^0). If need more, request a chuck for two elements (2^1), the next chuck requested will be of 4 elements (2^2).

The capacity of each chuck always is a power of two, and is related with the level. In the level 0, the capacity of the chuck is 2^0. In the level 1 the capacity id 2^1 and in the level 20 is 2^{20}

Each chuck, internally have a heap of bits for control the elements allocated, a counter of elements allocated and a vector with the memory to allocate. Each bit of the heap, control an element of the vector, as is described in the next point.

The heap of bits permit is a fast and easy way to know the first position not allocate in the chuck, and know when the chuck don't have any element allocated, and is empty



Each element deallocated in the pool, check if the last row is empty and if the number of elements allocated are the capacity of the pool divide by four **(NElem <= (NelemMax >>2).** If true, we can return to the allocator the last chuck, and decrease the size and the capacity of the pool

In the allocation, we obtain the first position not allocated from the heap, and return a pointer of that position in the vector, and set to 1 that position in the heap

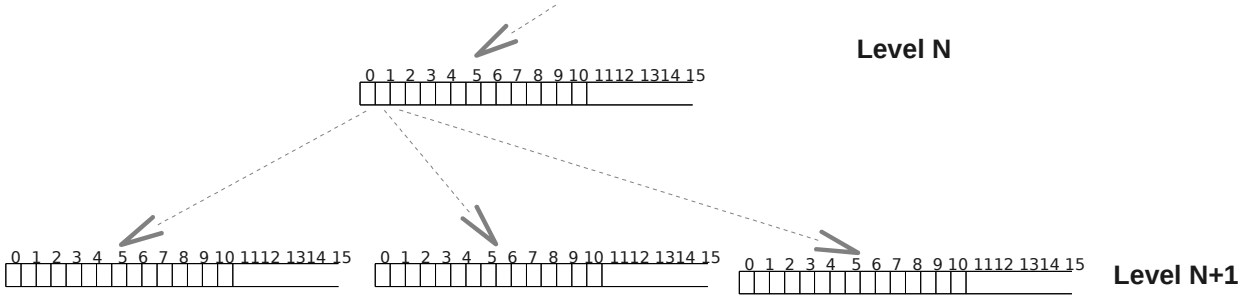
3.2.- THE HEAP

The heap is a tree of fixed size, with fixed relations between their elements. Due to this property, they don't need pointers or connectors for to connect their nodes. The elements of this heap are words of 64 bits.

The capacity of the heap always is a power of two and is fixed. If we create a heap with capacity for 4096 elements. You can allocate and deallocate elements, but the capacity can't change.

Each bit of the level N-1 control a word of the level N, and each bit of the level N control a word of the level N+ 1.

The bits of the last level control the positions allocated in the vector of memory, if the value is 1, that position had been allocated, if 0, the position is unused. In the upper level (level 0) there is only one word



Imagine we want create a heap for to control 2^{26} elements (67.108.864). We need 5 levels. The bits of the level 4 control the positions for to allocate in the vector of 2^{26} elements

Level	**Bits used in the level**	**Words used in the level**
0	16 (2^4)	1
1	1024 (2^{10})	16 (2^4)
2	65536 (2^{16})	1024 (2^{10})
3	1048576 (2^{20})	65536 (2^{16})
4	67108864 (2^{26})	1048576 (2^{20})

In the level 3, each word of the level 4 is controlled by a bit. Each word of 64 bits (2^6 bits) is controlled by a bit of the previous level. In the Level 4 we need 2^{26} bits, or ($2^{26} / 2^6 = 2^{20}$) words of 64 bits.

Repeating this calculation, at level 0, we need only 16 bits. We use the first 16 bits of a word of 64 bits. The others 48 bits are unused ad set to 1.

For the description of the algorithms, we use several variables

```
//---------------------------------------------------------------------
//  VARIABLES USED IN THE PSEUDOCODE OF THE HEAP
//---------------------------------------------------------------------
uint64_t const MAX64 ; // Word with all the bits set to 1
uint64_t const NBitMax ; // Number of elements controlled in the heap
uint64_t const NLevel ; // Number of levels in the heap
uint64_t *L[Nlevel] ; //vector of pointers to the first elements of each level
uint64_t NElem ; // Number of elements allocated in the heap
Element  * P ; // Pointer to the elements to allocate
```

The control of the words is :

- When a word is full (equal to MAX64), the bit which control it must be set to 1, in any other case is 0. And if you find a zero in a bit of the word, indicate that there are positions not allocated.
- When you set to 1 the bit which control the word, you must repeat the procedure, if this word is equal to MAX64, you must set to 1 the bit in the upper level, until you arrive to the level 0
- The operations for to know the corresponding word and bit in the upper level are done shifting 6 bits to the right, and doing a binary and with the number 63.
- For to know the word in the lower level we shift 6 bits to the left and add the bit position.

The code presented here is not the code of the class. It's a code with some simplifications on order to be clear. This code had not been compiled, it's only for to explain the algorithms.

```
void * allocate   ( void)
{   //------------------------ begin ----------------------------
    if ( L[0][0] == MAX64 ) return NULL ; //condition for to know if it is full
    //-------------------------------------------------------------------
    // If the only operations which modify the heap a reallocate and deallocate
    //-------------------------------------------------------------------------------
    uint64_t Level, Cursor ;
    for ( Cursor = 0, Level = 0 ; Level < NLevel ;++Level)
    {   Cursor= (Cursor << 6)+ FindFirstZero( L[Level][Cursor]);
    };
    //-------------------------------------------------------------------
    // set to one the bit
    //-------------------------------------------------------------------
    uint64_t N1 = Cursor , N2 = 0 ;
    Level = NLevel;
    uint64_t * PAux = NULL ;
    do
    {   N2 = N1 & 63 ;
        PAux = &(L[--Level][N1>>=6]);
        set_one64 ( *PAux, N2);
    } while ( (*PAux == MAX64) and Level != 0) ;
    NElem++;
    Return ( (void*) & P [Cursor]);
 };
```

```
bool  deallocate ( void * Ptr)
{   //------------------------ begin ----------------------------
    bool Found = ( & P[0]<= Ptr and P[NBitMax ]> Ptr );
    //-------------------------------------------------------------------
    // If Found is false, the pointer Ptr is not of this chuck
    //-------------------------------------------------------------------
    if  ( not Found )  return false;
    uint64_t Pos = Ptr - & P[0];

    uint64_t N1 = Pos , N2 = 0 , Level = NLevel;
    bool SW ;
    do
    {   N2 = N1 &63 ;
        uint64_t & Aux = L[--Level][N1 >>= 6];
        SW = ( Aux == MAX64 );
        set_zero64 ( Aux , N2);
    } while ( SW and Level != 0) ;
    NElem-- ;
    return true ;

};
```

3.3.-OTHERS ALGORITHMS

Some processors have a HW instruction for to find the first zero in a word. Obviously, this is the fastest method, but you can't use in all the processors.

The other option is by SW. This alternative is slower, but provide you the versatility of be used in any processor.

I had a function done by myself for to do this, but I found a little gem, a fast and elegant function using the DE Bruijn Sequences.

The code if from http://chessprogramming.wikispaces.com/BitScan#DeBruijnMultiplation The article where explain all is "Using DE Bruijn Sequences to Index 1 in a Computer Word". Martin Laüter, Charles E. Leiserson, Harald Prokop and Keith H. Randall (MIT 1997).

Really the algorithm find the first 1 in the word, and we passed the word complemented for to find the first zero.

```
const int index64[64] = {
   63,  0, 58,  1, 59, 47, 53,  2,
   60, 39, 48, 27, 54, 33, 42,  3,
   61, 51, 37, 40, 49, 18, 28, 20,
   55, 30, 34, 11, 43, 14, 22,  4,
   62, 57, 46, 52, 38, 26, 32, 41,
   50, 36, 17, 19, 29, 10, 13, 21,
   56, 45, 25, 31, 35, 16,  9, 12,
   44, 24, 15,  8, 23,  7,  6,  5
};

/**
 * bitScanForward
 * @author Martin Läuter (1997)
 *         Charles E. Leiserson
 *         Harald Prokop
 *         Keith H. Randall
 * "Using de Bruijn Sequences to Index a 1 in a Computer Word"
 * @param bb bitboard to scan
 * @precondition bb != 0
 * @return index (0..63) of least significant one bit
 */
int bitScanForward(U64 bb) {
   const U64 debruijn64 = C64(0x07EDD5E59A4E28C2);
   assert (bb != 0);
   return index64[((bb & -bb) * debruijn64) >> 58];
}
```