# Custom Memory Allocators

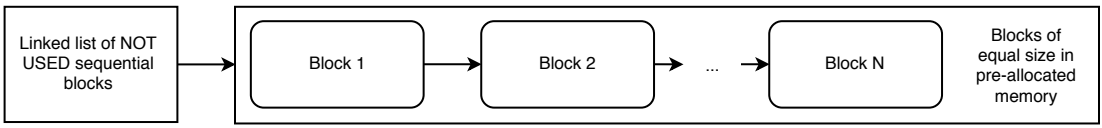
Features:

1. Defined model of allocation
2. No additional time to give the control to the OS
3. Do not have to initialize
4. Do not have to free
5. No memory leaks
6. Customizable for common task
7. Easy to debug

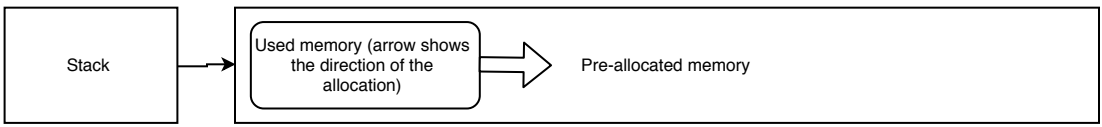
Examples:

1. Linked list of free blocks for each type of big enough resource
2. Stack
3. Double Stack
4. Single Frame Manager (memory for only one frame)
5. Double Buffer Manager (memory for current and next frames)

1:

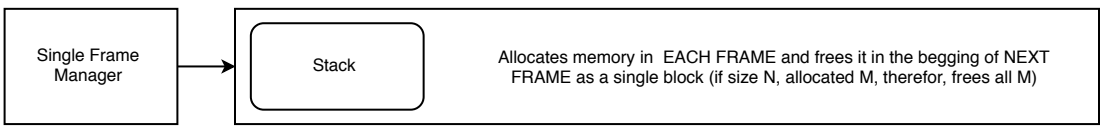



2:

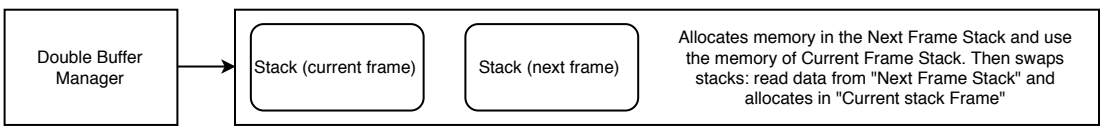



3:


4:




5:




Egor Orachyov, 2018

# Stack Manager

There is an example of StackManager's interface  for C:

```
typedef struct
{
VOID8 * mBuffer;
I64 mTop;
I64 mSize;
I64 mMarker;

} StackManager;

StackManager * StackManager_create(I64 size);
VOID8 * StackManager_delete(StackManager * sm);

VOID8 * StackManager_alloc(StackManager *sm, I64 bytes);
VOID8 * StackManager_calloc(StackManager * sm, I64 count, I64 size);

VOID8 StackManager_setMarker(StackManager * sm);
I64 StackManager_getMarker(StackManager * sm);
VOID8 StackManager_freeToMarker(StackManager * sm);
```

And  another C code, which shows us how we can compare performance of custom memory manager and standard C malloc.
We create in for loop dynamic arrays of int with fixed size and initialize in loop by same value.

Custommemory allocator

```
startTime = (double)clock();

for(int i = 0; i < count; i++) {
m1 = StackManager_alloc(manager, sizeof(int) * size);

for(int j = 0; j < size; j++) {
m1[j] = j;
}
};

full = (double)clock() - startTime;
printf("Custom allocator: %lf \n", full / count);
```

Custommemory allocator

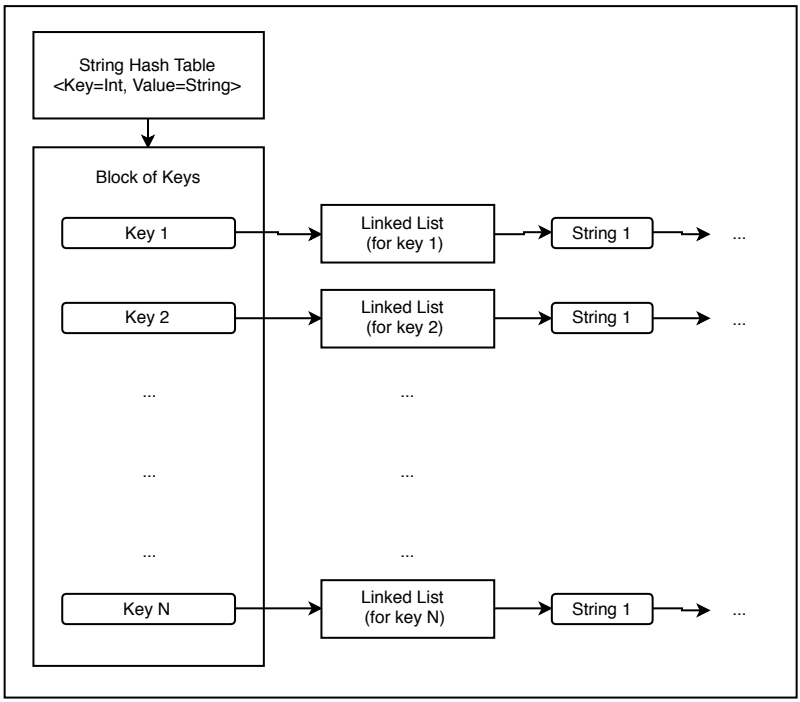
```
startTime = (double)clock();

for(int i = 0; i < count; i++) {
m2 = malloc(sizeof(int) * size);

for(int j = 0; j < size; j++) {
m2[j] = j;
}
};

full = (double)clock() - startTime;
printf("Standard allocator: %lf \n", full / count);
```

We get about 25% increase of speed for allocation
And more then 100% increase, if we have to free allocated memory for each frame (of course, if free
UNIT block of memory)

# Hashed Strings

To employ an enormous amount of strings in you program you could use hashed strings, what means:
- Global hash table for storing all allocated strings
- Using identifier for access to strings
- Comparing string for O(1) in average by integer hashes
- Ability to free all program strings' memory by simple deleting of Hash Table

Here some example of how it could be implemented:




And an example of interface to interact with strings:

```
cString * getString(I32 stringId);
bool * isStringLoaded(cString * str);
I32 loadStringInTable(Cstring * str);
```

An example of usage:

```
I32 sid_helloWorld = loadStringInTable("Hello, world!");
fprintf(stdout, "%s\n", getString(sid_helloWorld));
```

Egor Orachyov, 2018