

성능을 위한 메모리 사용 tip

유영천

<https://megayuchi.com>

tw:@dgtman

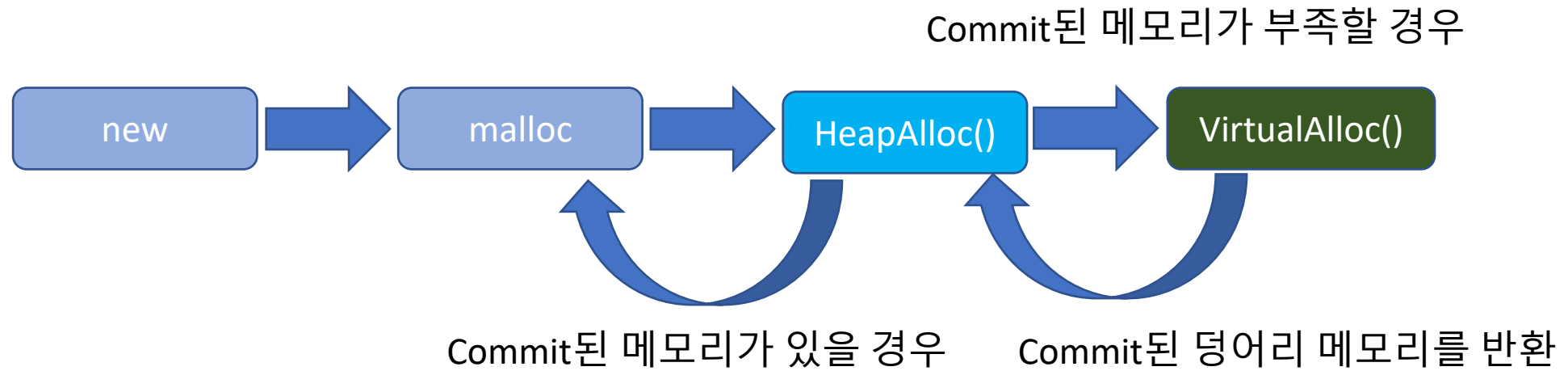
메모리 관리 기본

- OS의 메모리 매니저의 기본 관리 단위는 페이지(4KB / 1MB / 2MB / 4MB)
- 1Byte메모리 할당에 4KB를 쓸수는 없다.
- 임의의 덩어리 메모리를 쪼개서 다양한 사이즈의 메모리 블록으로 사용할 수 있도록 하자. -> Heap

Windows의 메모리 할당

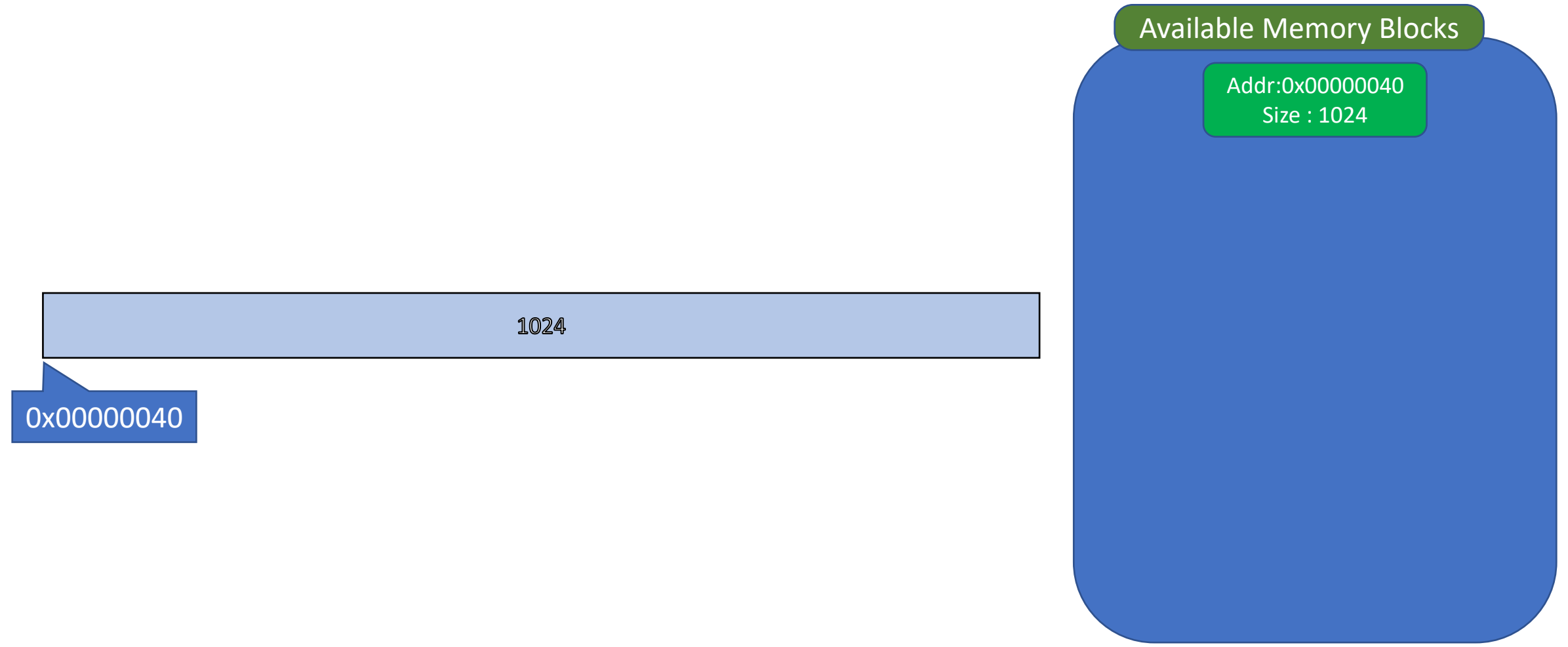
- new/malloc -> HeapAlloc() -> VirtualAlloc()

Windows의 메모리 할당

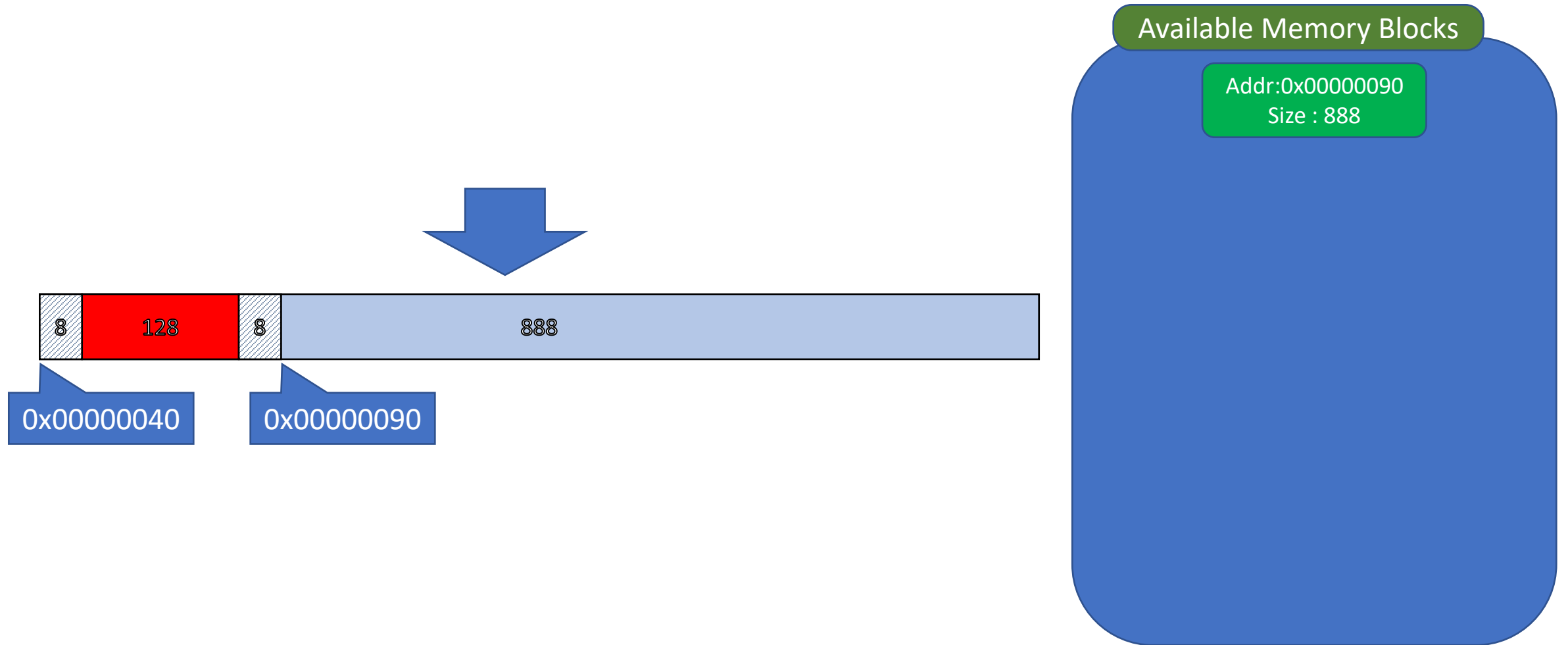


Heap 메모리

- 큰 덩어리의 메모리부터 다양한 사이즈의 메모리 블록을 할당.
- 해제 시 인접한 블록과 병합할 수 있으면 병합하여 더 큰 메모리 블록을 유지.
- 대부분의 메모리 관리 시스템이 heap based.

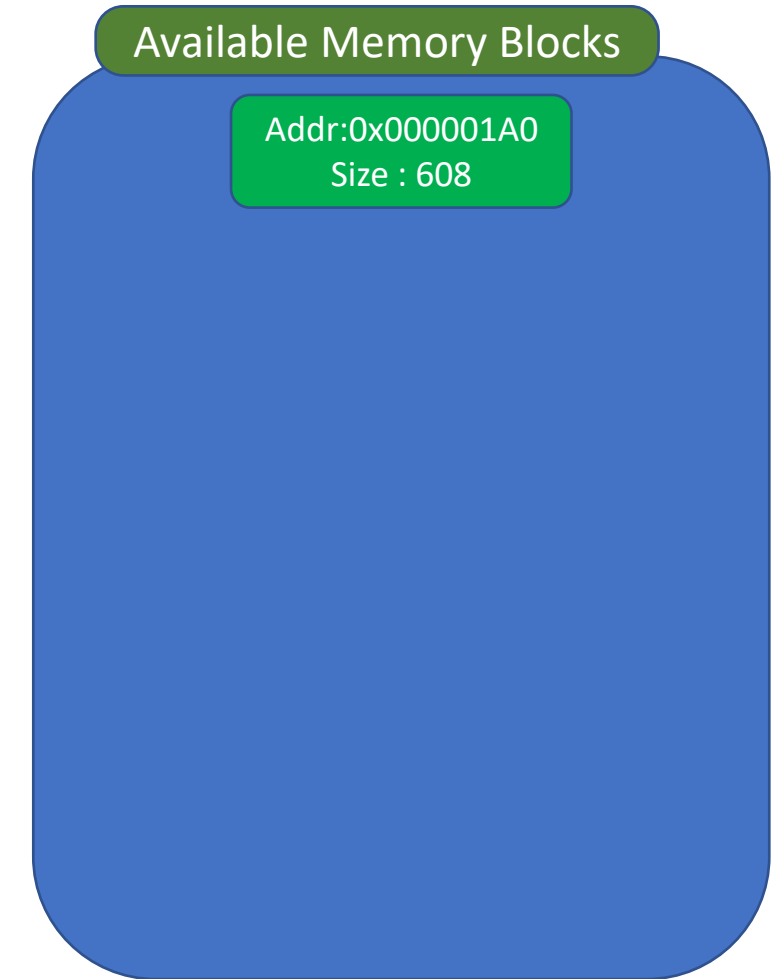
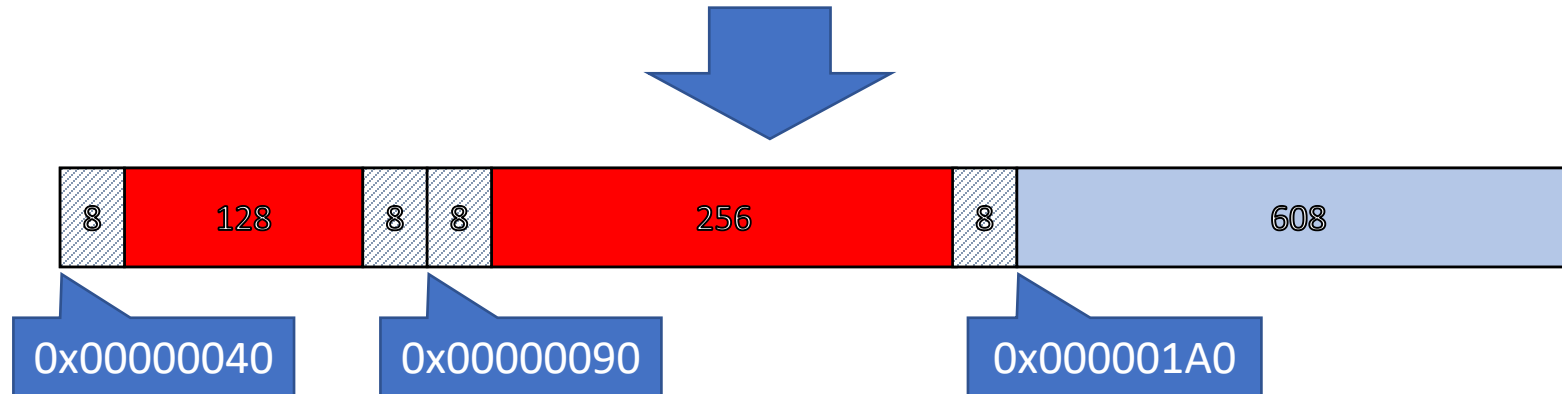


Alloc(128) -> ret (0x00000040 + head(8))



Alloc(128) -> ret (0x00000040 + head(8))

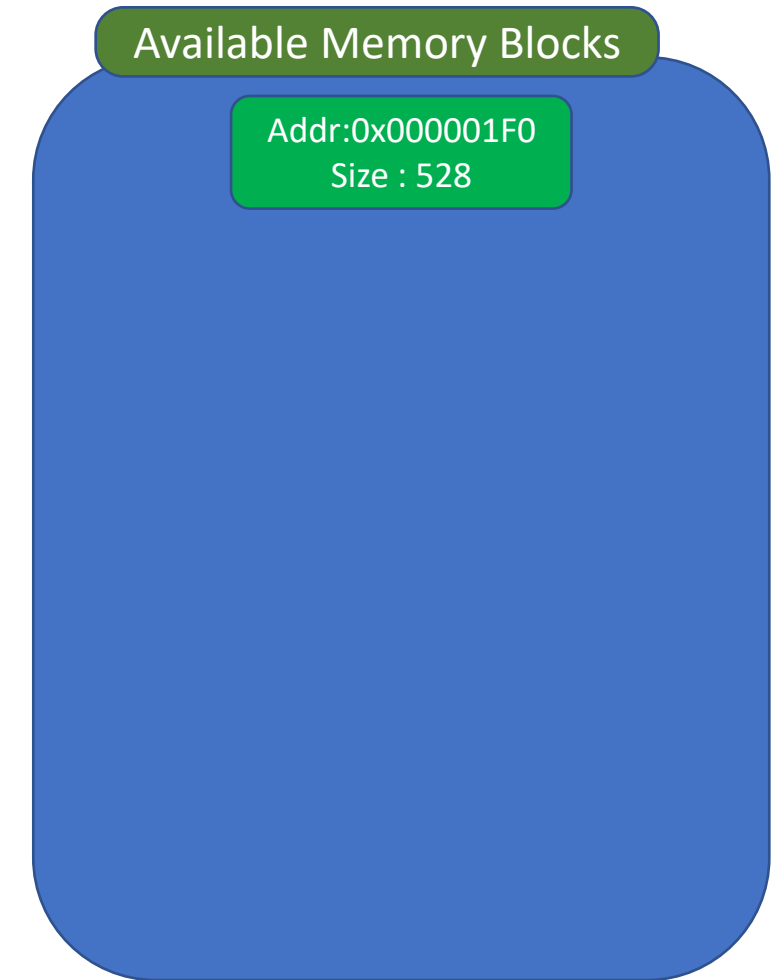
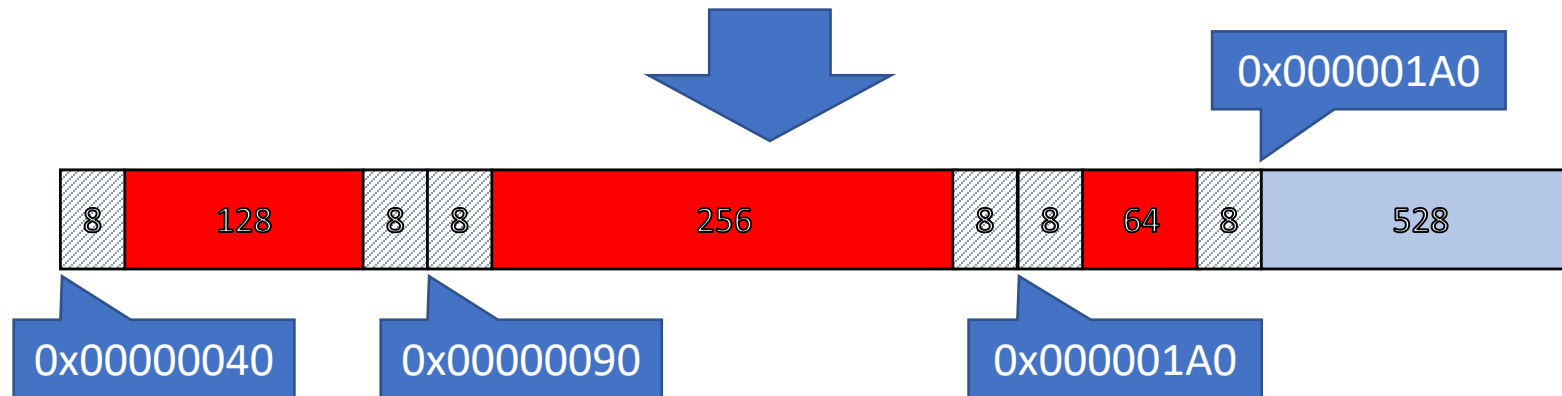
Alloc(256) -> ret (0x00000090 + head(8))



Alloc(128) -> ret (0x00000040 + head(8))

Alloc(256) -> ret (0x00000090 + head(8))

Alloc(64) -> ret (0x000001A0 + head(8))

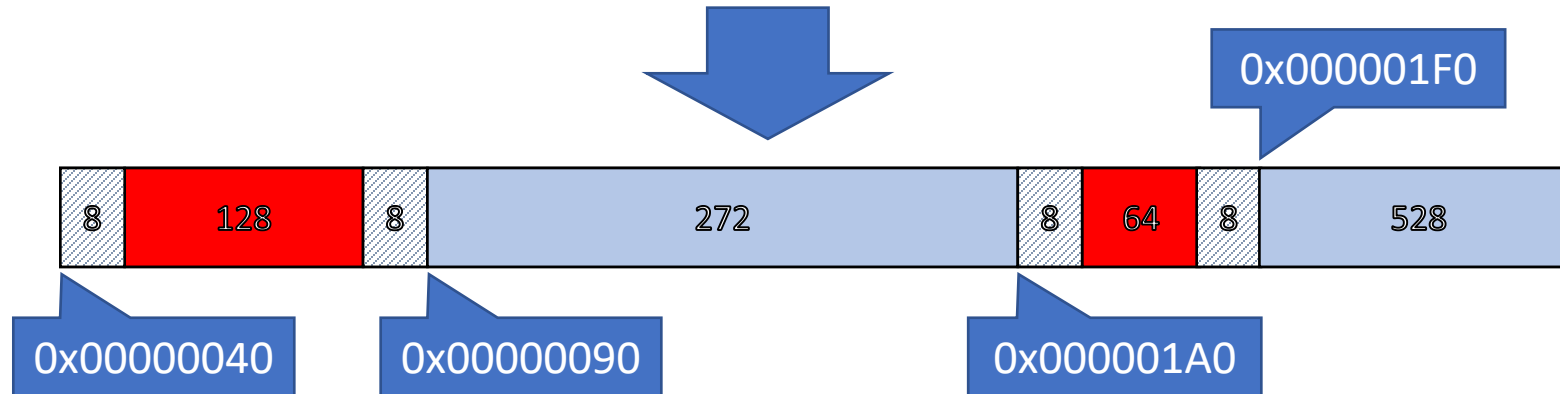


Alloc(128) -> ret (0x00000040 + head(8))

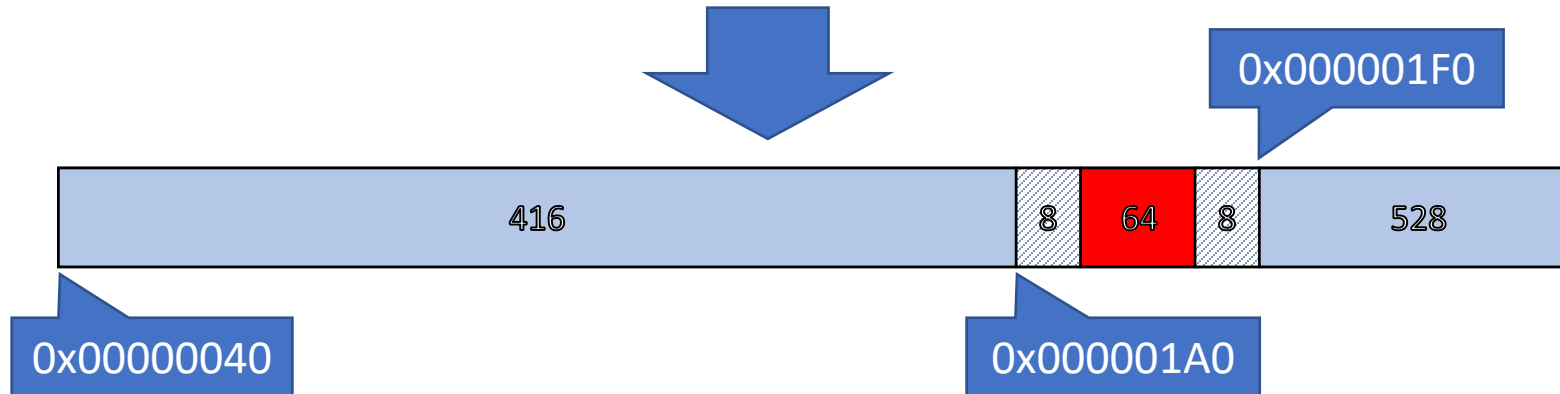
Alloc(256) -> ret (0x00000090 + head(8))

Alloc(64) -> ret (0x000001A0 + head(8))

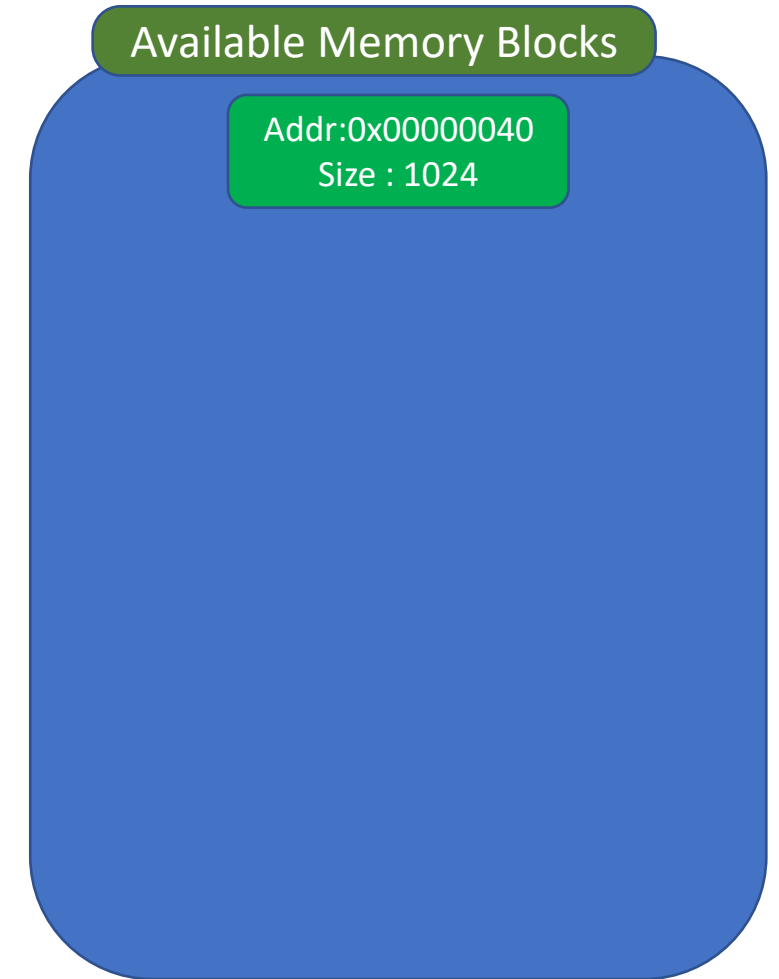
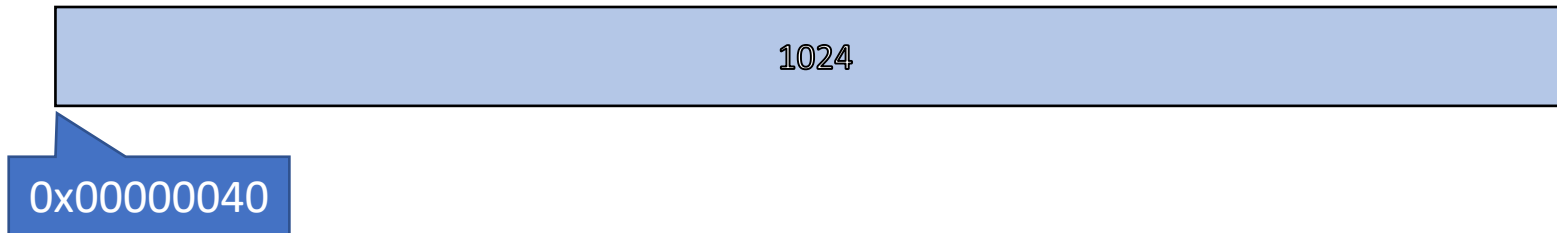
Free(0x00000090 + 8) -> 병합불가



Alloc(128) -> ret (0x00000040 + head(8))
Alloc(256) -> ret (0x00000090 + head(8))
Alloc(64) -> ret (0x000001A0 + head(8))
Free(0x00000090 + 8) -> 병합불가
Free(0x00000040 + 8) -> 0x00000090 블록과 병합가능



Alloc(128) -> ret (0x00000040 + head(8))
Alloc(256) -> ret (0x00000090 + head(8))
Alloc(64) -> ret (0x000001A0 + head(8))
Free(0x00000090 + 8) -> 병합불가
Free(0x00000040 + 8) -> 0x00000090 블록과 병합가능
Free(0x000001A0 + 8) -> 0x000001F0 블록과 병합가능



Heap의 성능 문제

- 적합한 사이즈의 블록 찾기
- 해제 시 병합 비용
- 단편화
- Commit 비용

메모리 사용 특성 고려

- 특정 사이즈(struct의 사이즈)의 메모리 블록을 왕창 할당할 일이 많다.
- 게임에서는 메모리 사용량 peak에 도달한 후 다시 peak에 도달할 가능성이 높다(굳이 즉시 해제할 필요가 없다).

고정 사이즈 메모리풀

- 사이즈가 고정된다.
 - 적합한 사이즈의 메모리를 탐색할 필요가 없다.
- 병합이 필요 없다.
- 단편화가 생기지 않는다.
- 다양한 사이즈의 메모리를 할당할 수는 없으므로 heap을 완전히 대체할 수는 없다.

Index = block ptr – base ptr



32	32	32	32	32	32	32	32	32	32	32	32	32	32	32	32
(0)	(1)	(2)	(3)	(4)	(5)	(6)	(7)	(8)	(9)	(10)	(11)	(12)	(13)	(14)	(15)

0x00000040 (base ptr)

Available Memory Blocks

Pointer/index table

0	1	2	3
4	5	6	7
8	9	10	11
12	13	14	15

Alloc() -> ret(0x00000040)



Index = block ptr – base ptr

32 (0)	32 (1)	32 (2)	32 (3)	32 (4)	32 (5)	32 (6)	32 (7)	32 (8)	32 (9)	32 (10)	32 (11)	32 (12)	32 (13)	32 (14)	32 (15)
-----------	-----------	-----------	-----------	-----------	-----------	-----------	-----------	-----------	-----------	------------	------------	------------	------------	------------	------------

0x00000040 (base ptr)

Available Memory Blocks


Pointer/index table

	1	2	3
4	5	6	7
8	9	10	11
12	13	14	15

Alloc() -> ret(0x00000040)

Alloc() -> ret(0x00000060)

Index = block ptr – base ptr



32	32	32	32	32	32	32	32	32	32	32	32	32	32	32	32
(0)	(1)	(2)	(3)	(4)	(5)	(6)	(7)	(8)	(9)	(10)	(11)	(12)	(13)	(14)	(15)

0x00000040 (base ptr)

Available Memory Blocks

Pointer/index table

		2	3
4	5	6	7
8	9	10	11
12	13	14	15

Alloc() -> ret(0x00000040)
Alloc() -> ret(0x00000060)
Alloc() -> ret(0x00000080)



Index = block ptr – base ptr

32	32	32	32	32	32	32	32	32	32	32	32	32	32	32	32
(0)	(1)	(2)	(3)	(4)	(5)	(6)	(7)	(8)	(9)	(10)	(11)	(12)	(13)	(14)	(15)

0x00000040 (base ptr)

Available Memory Blocks

Pointer/index table

			3
4	5	6	7
8	9	10	11
12	13	14	15

Alloc() -> ret(0x00000040)
Alloc() -> ret(0x00000060)
Alloc() -> ret(0x00000080)
Free(0x00000060)



Index = block ptr – base ptr

32	32	32	32	32	32	32	32	32	32	32	32	32	32	32	32
(0)	(1)	(2)	(3)	(4)	(5)	(6)	(7)	(8)	(9)	(10)	(11)	(12)	(13)	(14)	(15)

0x00000040 (base ptr)

Available Memory Blocks

Pointer/index table

		1	3
4	5	6	7
8	9	10	11
12	13	14	15

Alloc() -> ret(0x00000040)
Alloc() -> ret(0x00000060)
Alloc() -> ret(0x00000080)
Free(0x00000060)
Free(0x00000040)



Index = block ptr – base ptr

32	32	32	32	32	32	32	32	32	32	32	32	32	32	32	32
(0)	(1)	(2)	(3)	(4)	(5)	(6)	(7)	(8)	(9)	(10)	(11)	(12)	(13)	(14)	(15)

0x00000040 (base ptr)

Available Memory Blocks

Pointer/index table

	0	1	3
4	5	6	7
8	9	10	11
12	13	14	15

Alloc() -> ret(0x00000040)
Alloc() -> ret(0x00000060)
Alloc() -> ret(0x00000080)
Free(0x00000060)
Free(0x00000040)
Free(0x00000080)



Index = block ptr – base ptr

32	32	32	32	32	32	32	32	32	32	32	32	32	32	32	32
(0)	(1)	(2)	(3)	(4)	(5)	(6)	(7)	(8)	(9)	(10)	(11)	(12)	(13)	(14)	(15)

0x00000040 (base ptr)

Available Memory Blocks

Pointer/index table

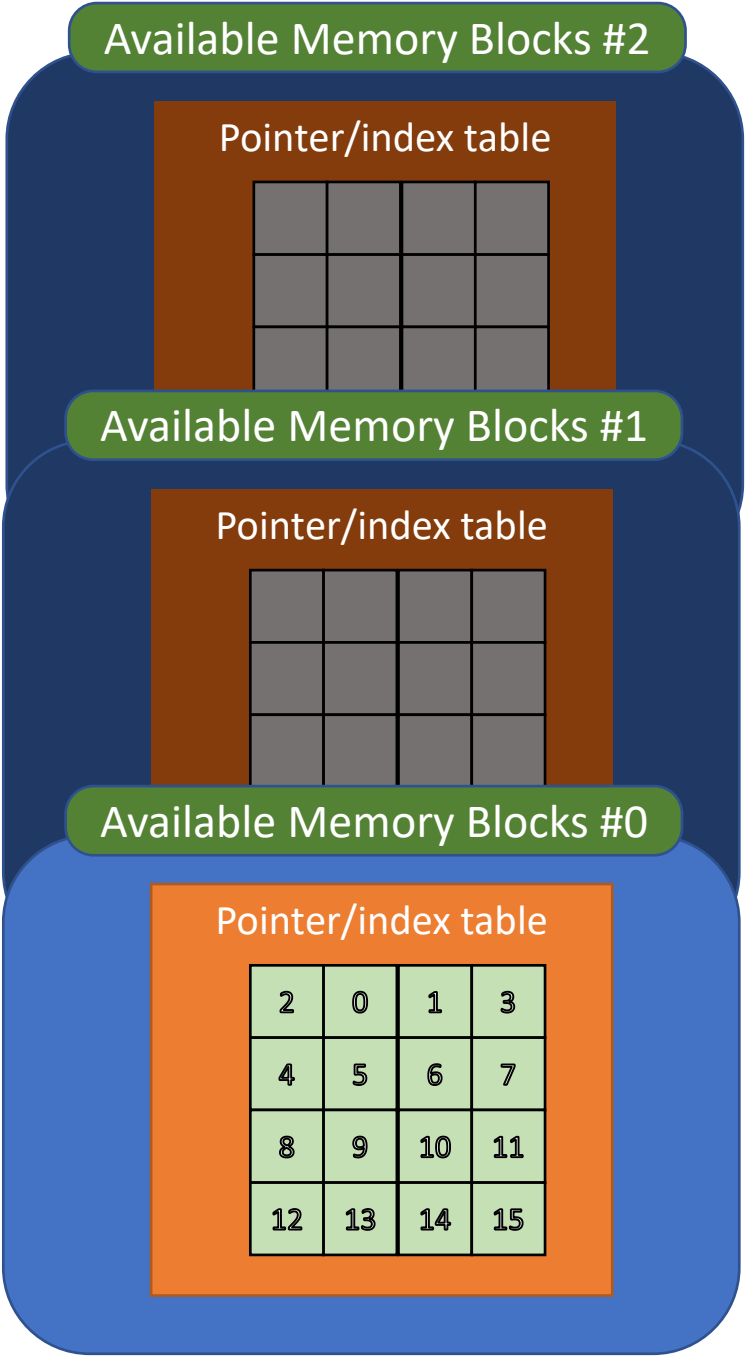
2	0	1	3
4	5	6	7
8	9	10	11
12	13	14	15

조금 더 똑똑하게

- 최대 블록 개수만큼 모두 할당해둘 필요는 없다.
- 최대 개수는 넉넉하게. 기본 할당 개수는 적게.

```
for (int i=0; i<16; i++)
{
    p[i] = Alloc();
}
```

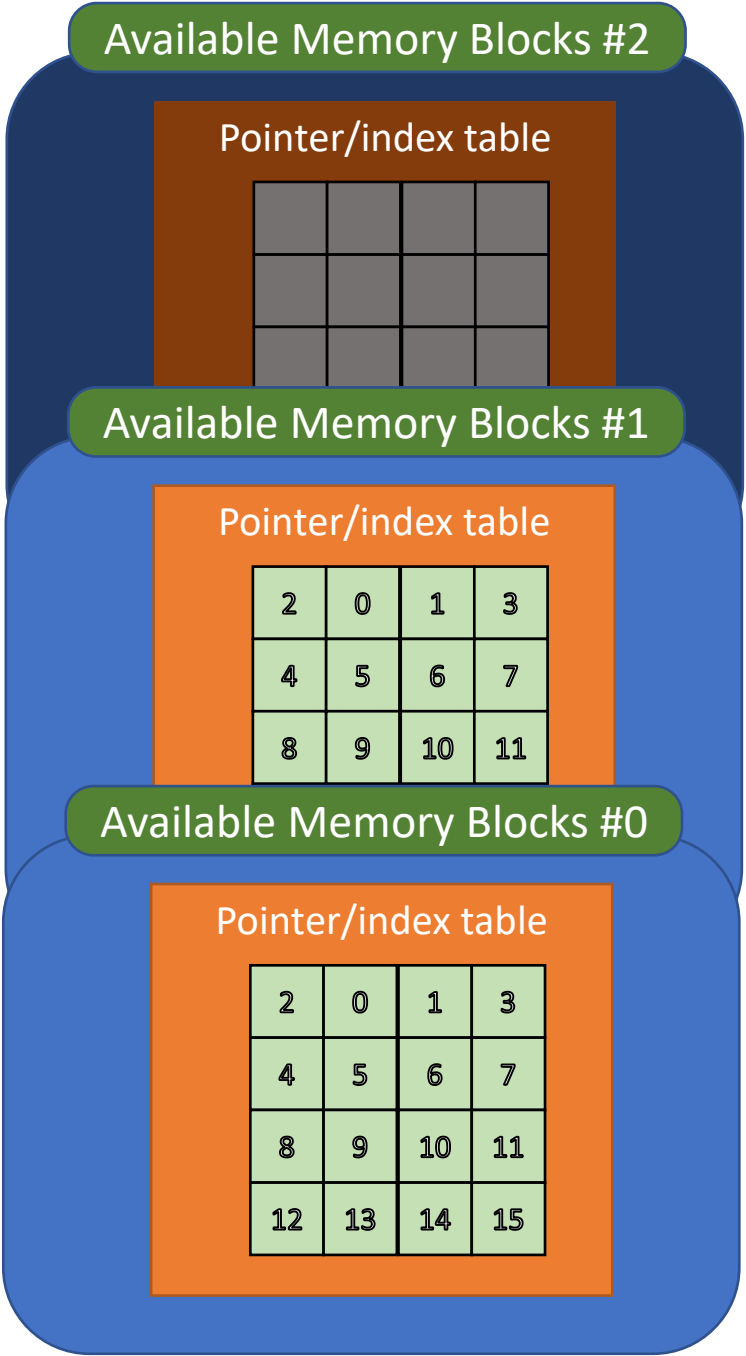
0	32 (0)	0	32 (1)	0	32 (2)	0	32 (3)	0	32 (4)	0	32 (5)	0	32 (6)	0	32 (7)	0
0	32 (8)	0	32 (9)	0	32 (10)	0	32 (11)	0	32 (12)	0	32 (13)	0	32 (14)	0	32 (15)	0




```
for (int i=0; i<32; i++)
{
    p[i] = Alloc();
}
```

1	32 (0)	1	32 (1)	1	32 (2)	1	32 (3)	1	32 (4)	1	32 (5)	1	32 (6)	1	32 (7)	1
1	32 (8)	1	32 (9)	1	32 (10)	1	32 (11)	1	32 (12)	1	32 (13)	1	32 (14)	1	32 (15)	1

0	32 (0)	0	32 (1)	0	32 (2)	0	32 (3)	0	32 (4)	0	32 (5)	0	32 (6)	0	32 (7)	0
0	32 (8)	0	32 (9)	0	32 (10)	0	32 (11)	0	32 (12)	0	32 (13)	0	32 (14)	0	32 (15)	0

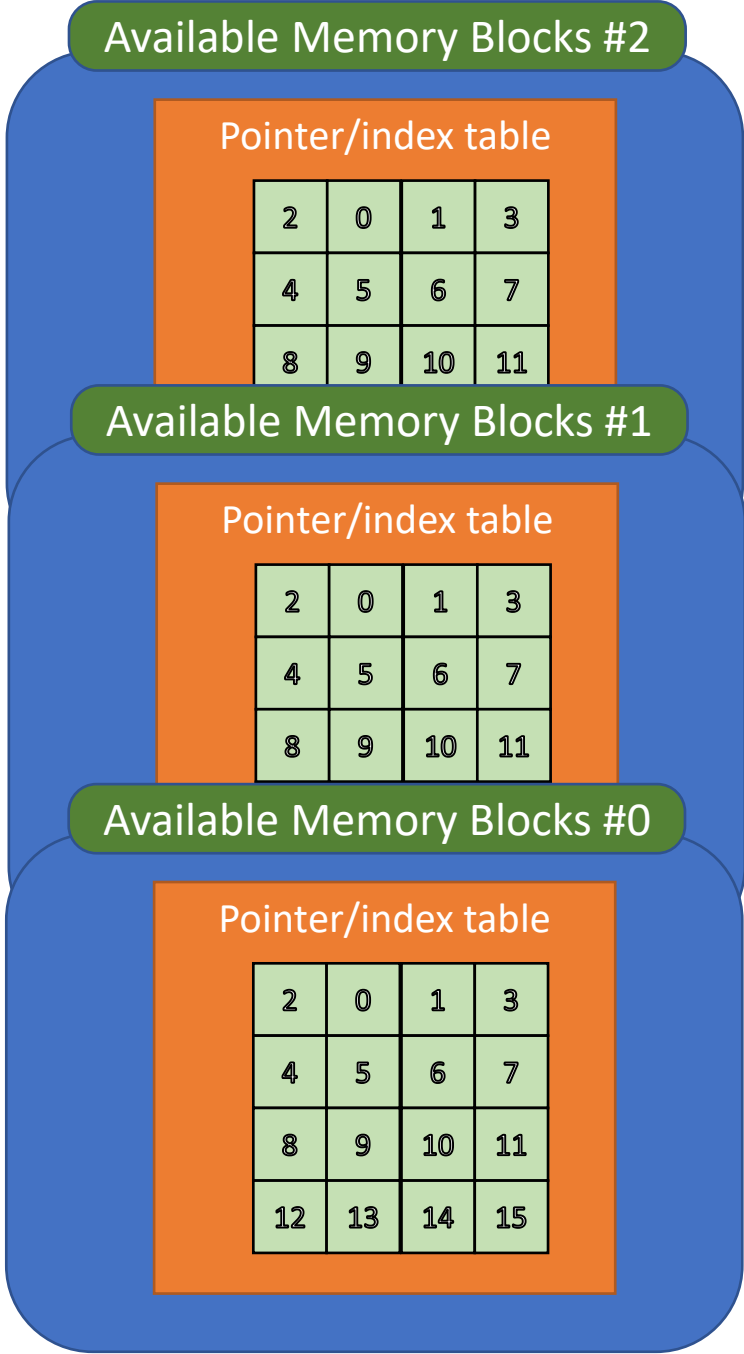


```
for (int i=0; i<48; i++)
{
    p[i] = Alloc();
}
```

2	32 (0)	2	32 (1)	2	32 (2)	2	32 (3)	2	32 (4)	2	32 (5)	2	32 (6)	2	32 (7)	2
2	32 (8)	2	32 (9)	2	32 (10)	2	32 (11)	2	32 (12)	2	32 (13)	2	32 (14)	2	32 (15)	2

1	32 (0)	1	32 (1)	1	32 (2)	1	32 (3)	1	32 (4)	1	32 (5)	1	32 (6)	1	32 (7)	1
1	32 (8)	1	32 (9)	1	32 (10)	1	32 (11)	1	32 (12)	1	32 (13)	1	32 (14)	1	32 (15)	1

0	32 (0)	0	32 (1)	0	32 (2)	0	32 (3)	0	32 (4)	0	32 (5)	0	32 (6)	0	32 (7)	0
0	32 (8)	0	32 (9)	0	32 (10)	0	32 (11)	0	32 (12)	0	32 (13)	0	32 (14)	0	32 (15)	0



성능 테스트

- 고정 사이즈 메모리 풀 vs CRT Heap
- 2^n 사이즈 메모리 풀 vs CRT Heap

Fixed Size
size = 256 Bytes
count = 65536

	Alloc	Free
Static Memory Pool	0.62	2.03
CRT Heap	8.14	5.33
	+1312%	+262%

Fixed Size
size = 256 KBytes
count = 65536

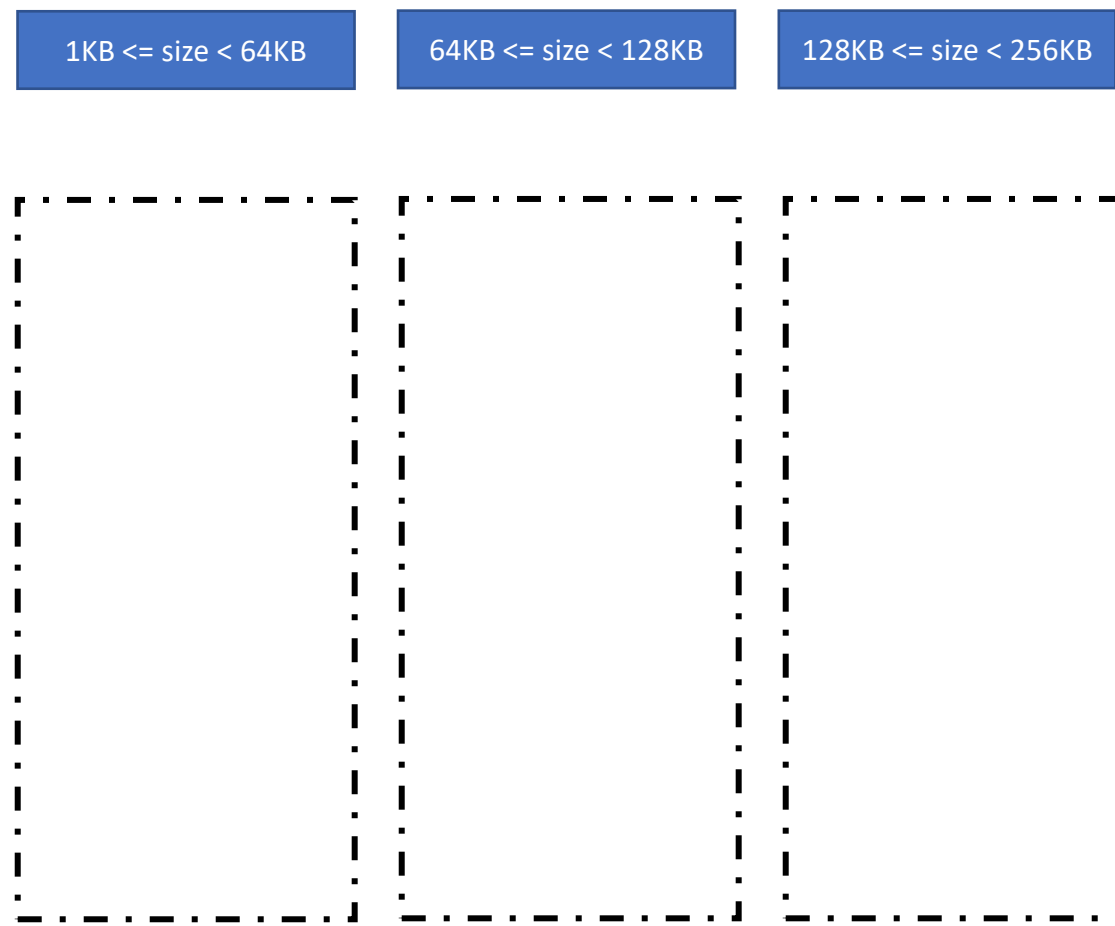
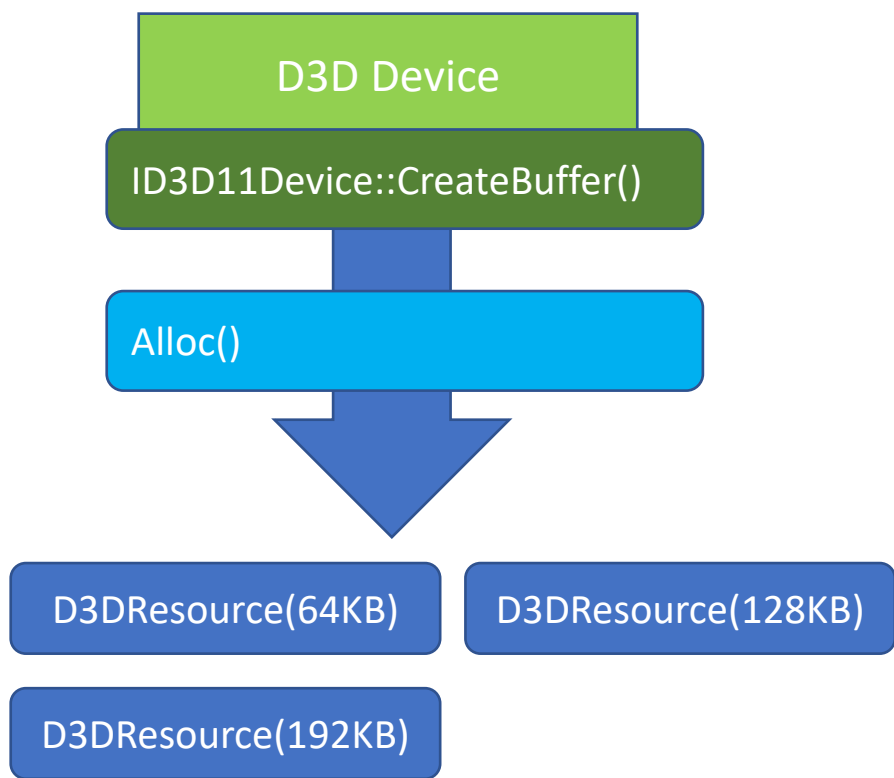
	Alloc	Free
Static Memory Pool	2.29	8.48
CRT Heap	254.17	939.41
	+11099%	+10026%

variable size
size=(1 ~ 1024) bytes
count = 65536

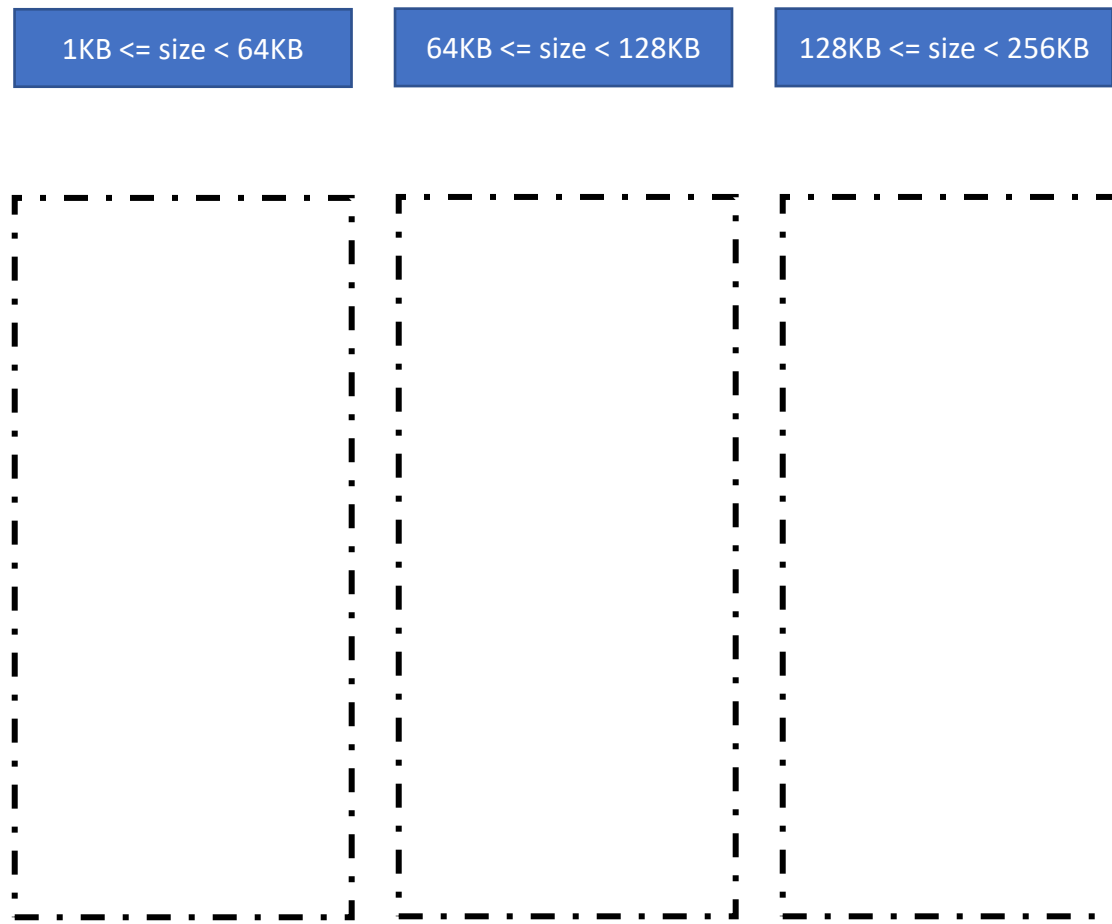
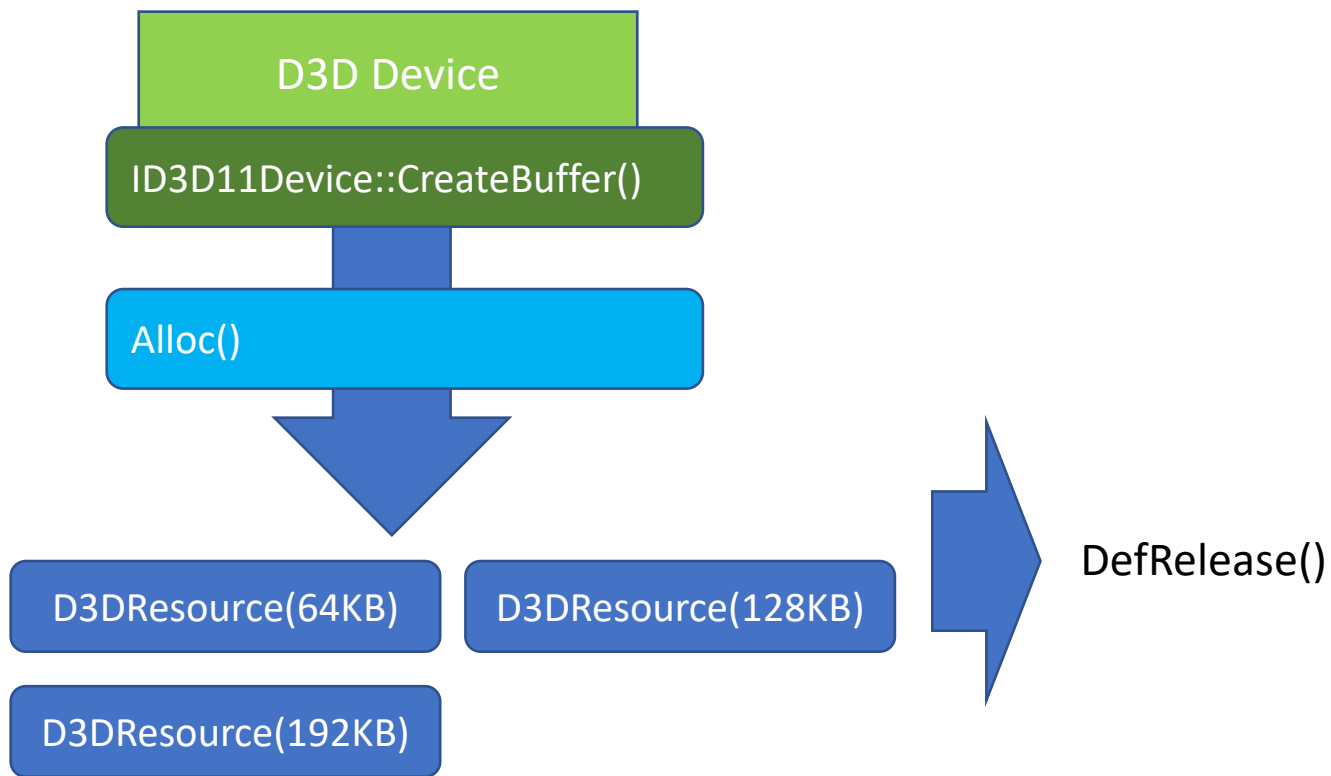
	Alloc	Free
Static Memory Pool(Pow(2))	6.51	2.06
CRT Heap	16.89	9.22
	+259%	+447%

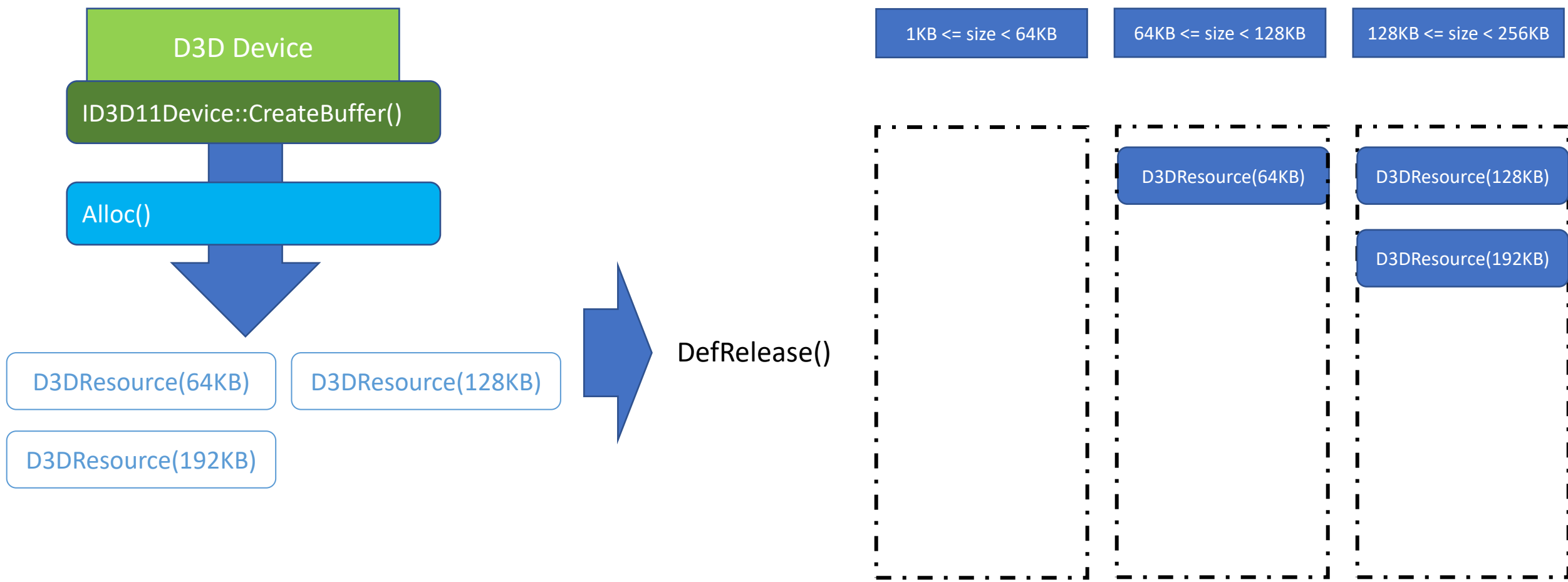
메모리 블록 재활용

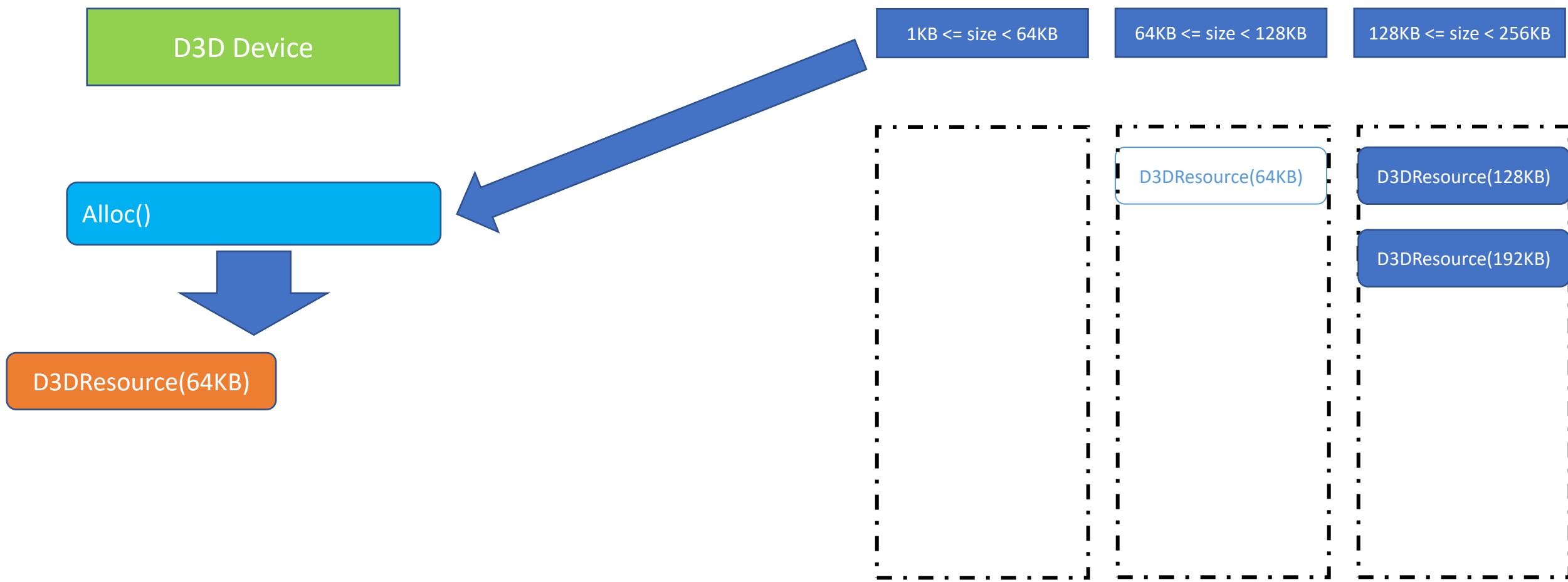
- CRT heap 또는 API에서 제공하는 할당 방법을 사용하되 한번 할당한 메모리(리소스)는 재활용한다.
- 해제 타이밍에 즉시 해제하지 않고 사이즈별로 분류해서 재활용 리스트에 등록해 둔다.
- 할당 타이밍에 재활용 리스트를 먼저 조회해서 적합한 사이즈의 메모리가 있으면 재활용한다.
- D3D등 정해진 방법으로 리소스를 할당해야 할 경우 유용하다.



메모리 블록 재활용







여러 번 할당, 한번에 해제

- 특정 타이밍에 여러 개의 블록을 할당하고 지속적으로 사용되지 않는 경우.
- 선형 메모리로부터 순차적으로 할당(포인터 offset만 증가)한다.
- 더 이상 어떤 메모리도 사용되지 않을 때 선형 메모리를 통째로 해제한다.
- Tree빌드 등에 유용하다.

Linear Memory



```
Begin()  
{  
    InitMemoryPool(1024);  
}
```

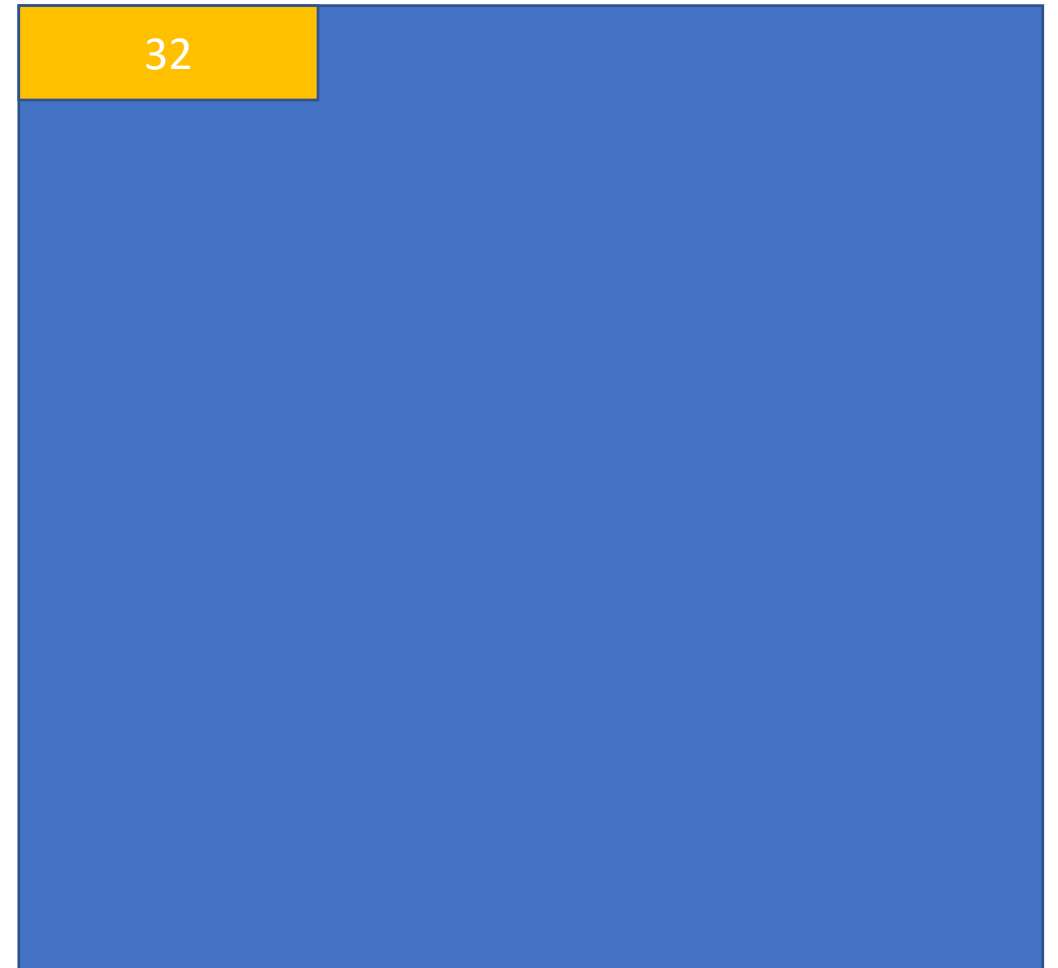
Linear Memory



```
Begin()  
{  
    InitMemoryPool();  
}
```

```
Build()  
{  
    Alloc(32);
```

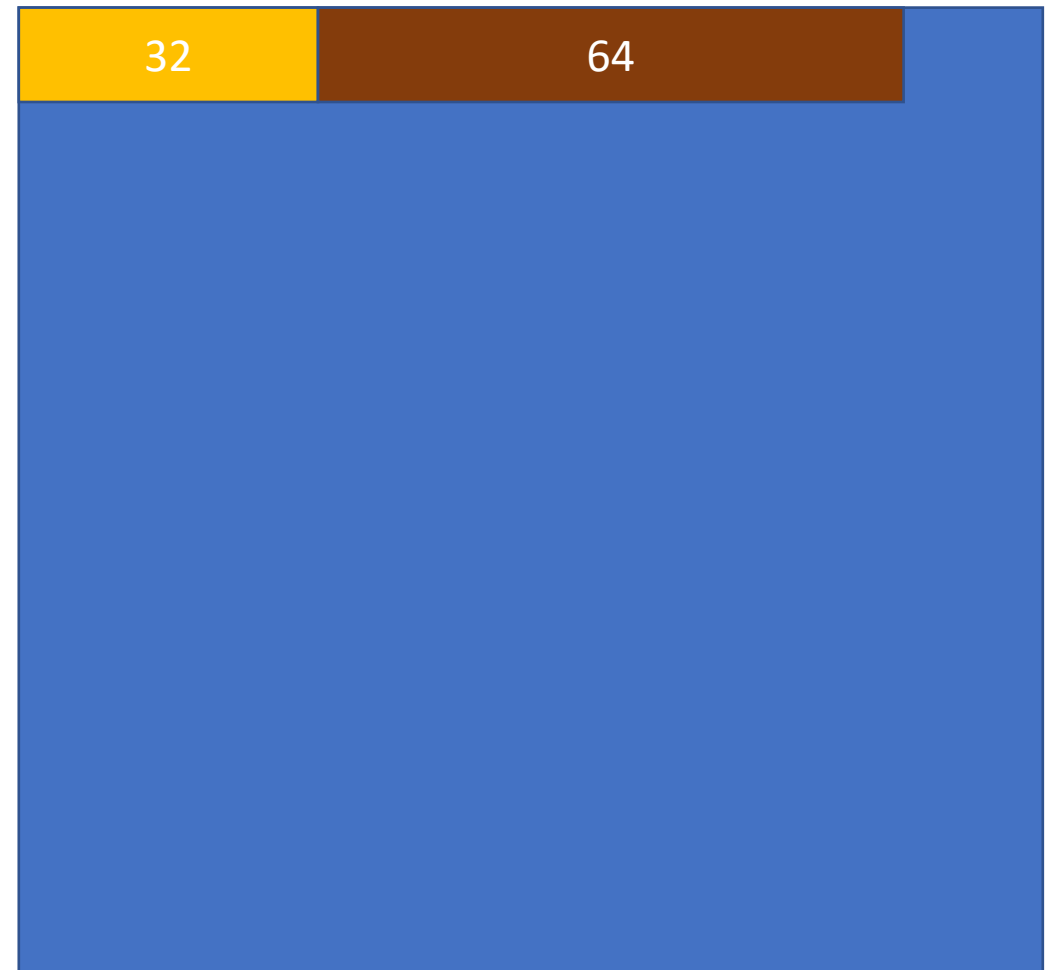
Linear Memory



```
Begin()  
{  
  InitMemoryPool();  
}
```

```
Build()  
{  
  Alloc(32);  
  Alloc(64);  
}
```

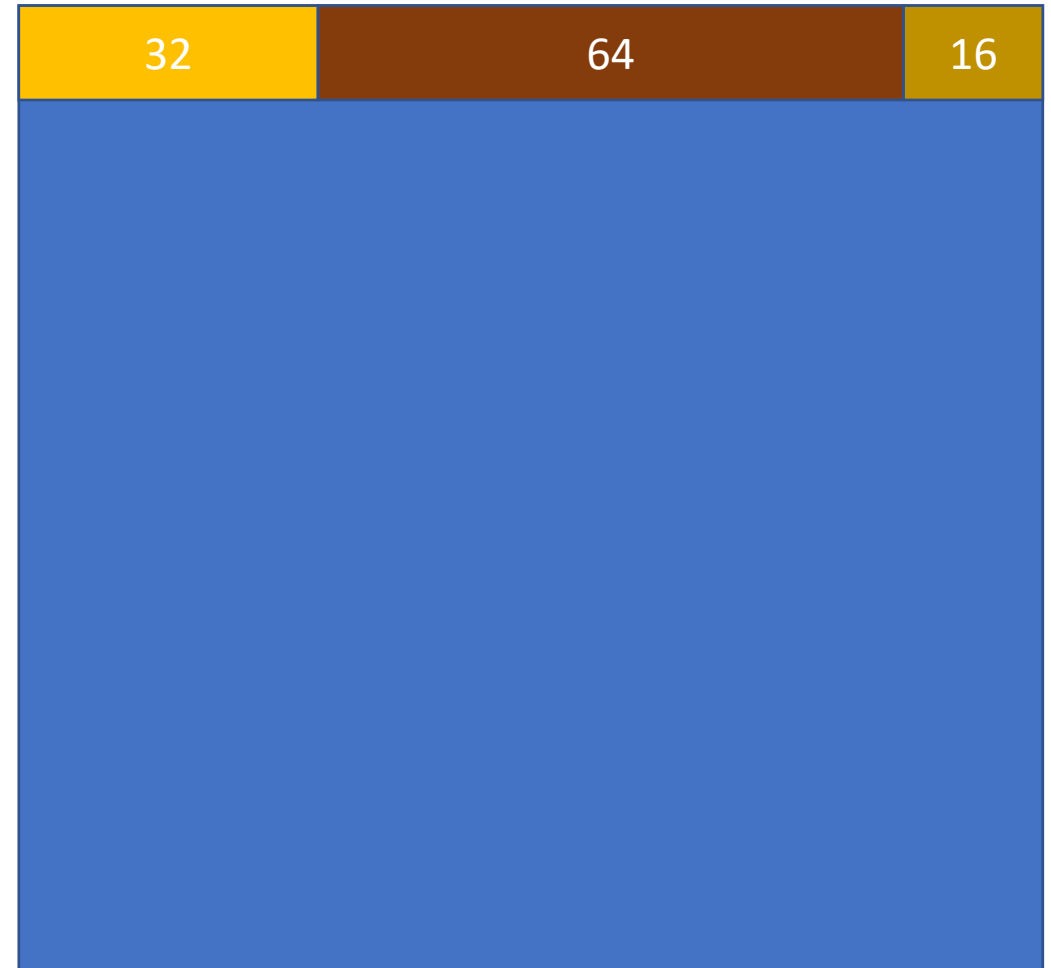
Linear Memory



```
Begin()  
{  
  InitMemoryPool();  
}
```

```
Build()  
{  
  Alloc(32);  
  Alloc(64);  
  Alloc(16);  
}
```

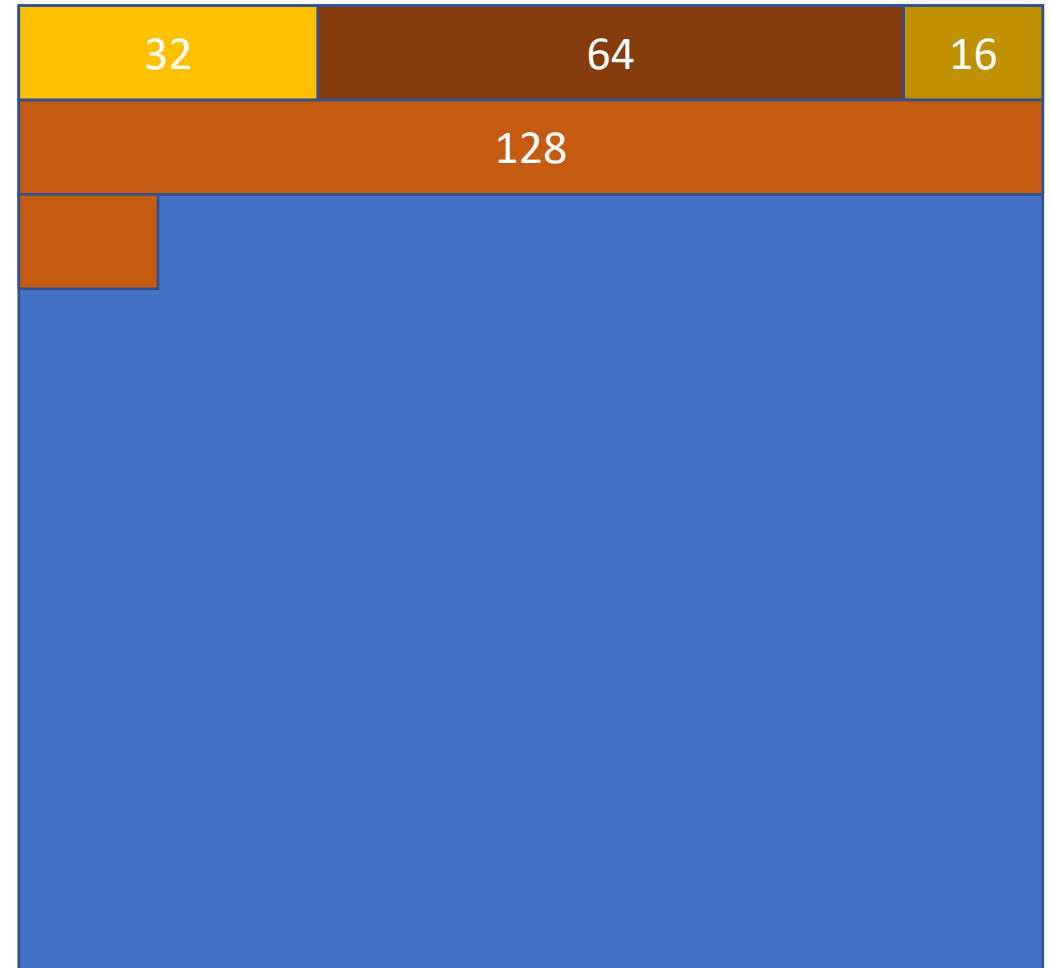
Linear Memory



```
Begin()  
{  
    InitMemoryPool();  
}
```

```
Build()  
{  
    Alloc(32);  
    Alloc(64);  
    Alloc(16);  
    Alloc(128);  
}
```

Linear Memory




```
Begin()
{
    InitMemoryPool();
}

Build()
{
    Alloc(32);
    Alloc(64);
    Alloc(16);
    Alloc(128);
}

End()
{
    ResetMemoryPool();
}

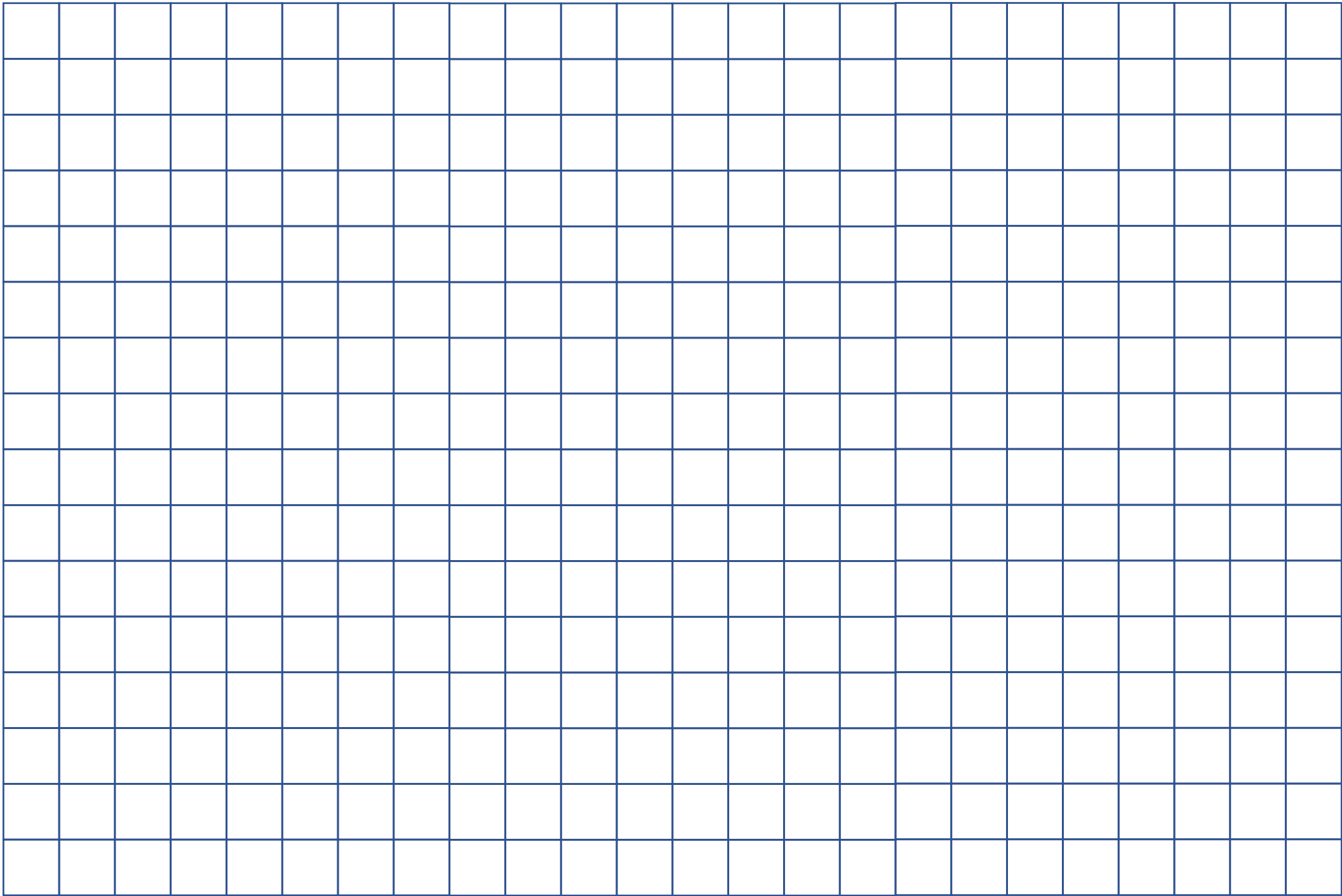
Cleanup()
{
    CleanupMemoryPool();
}
```

Linear Memory



On demand 배열

- 거대한 배열이 필요할 때
 - 그러나 그 배열의 일부만 사용되고
 - 어느 영역이 사용될지 알 수 없다.
- NxNxN씩 잘라서 배열 그룹을 만든다.
- 각 배열그룹 내의 메모리는 할당하지 않는다
- Access(), Alloc()등의 함수로 요청이 들어오면 그때 할당하고 포인터를 돌려준다.
- Paging기법과 유사
- 월드 공간을 그리드로 구현할 때 유용함.



Sector (0,0)

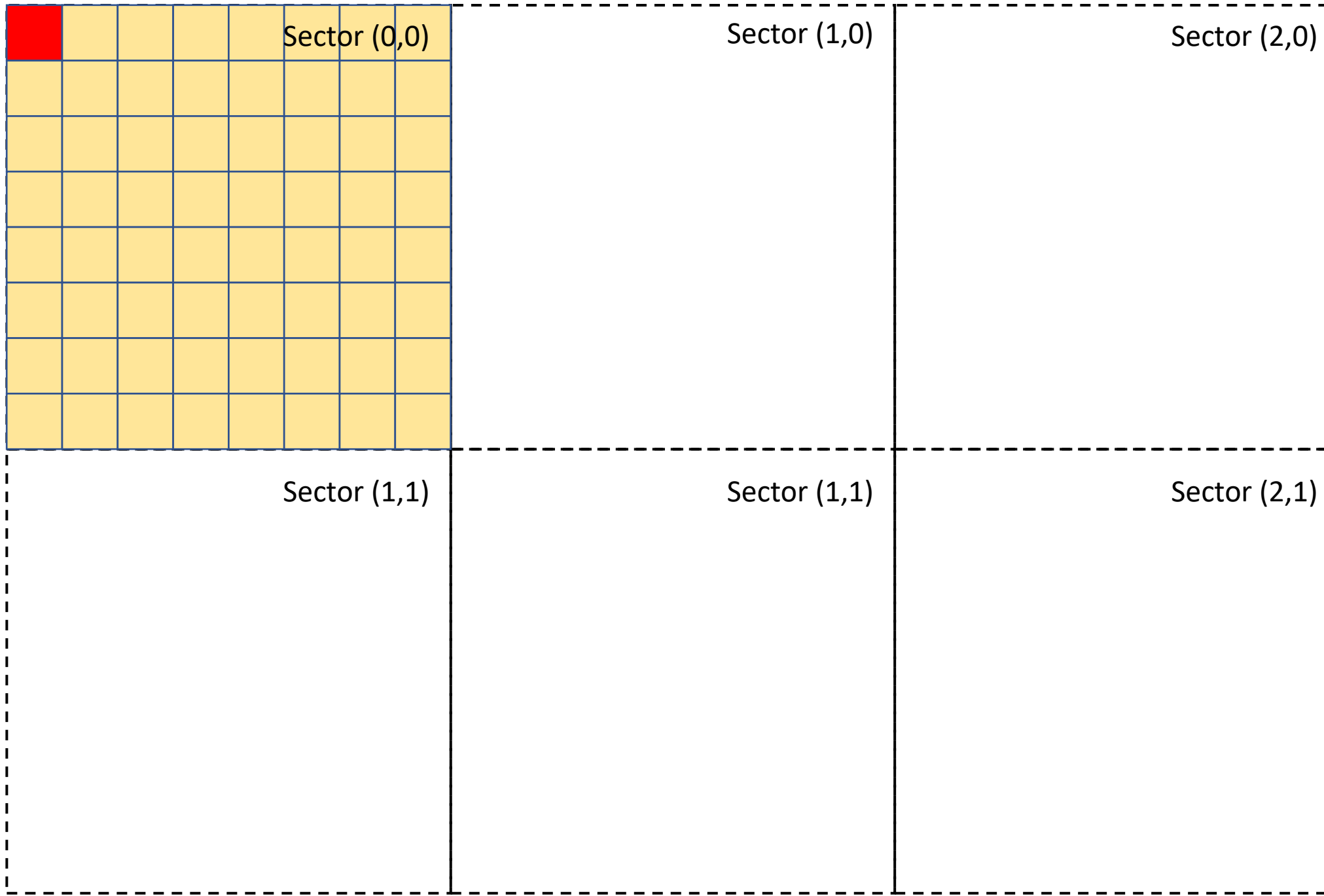
Sector (1,0)

Sector (2,0)

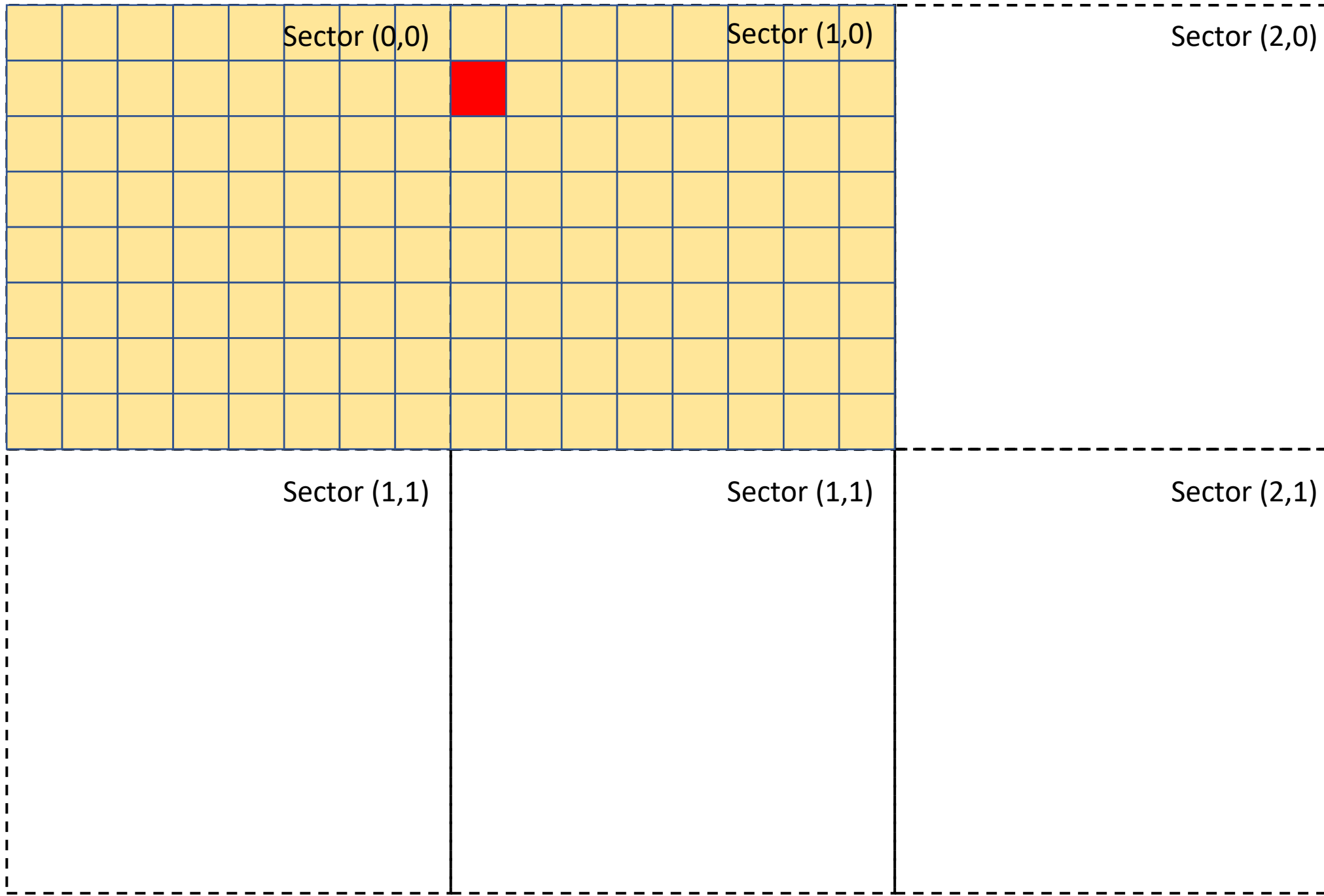
Sector (1,1)

Sector (1,1)

Sector (2,1)

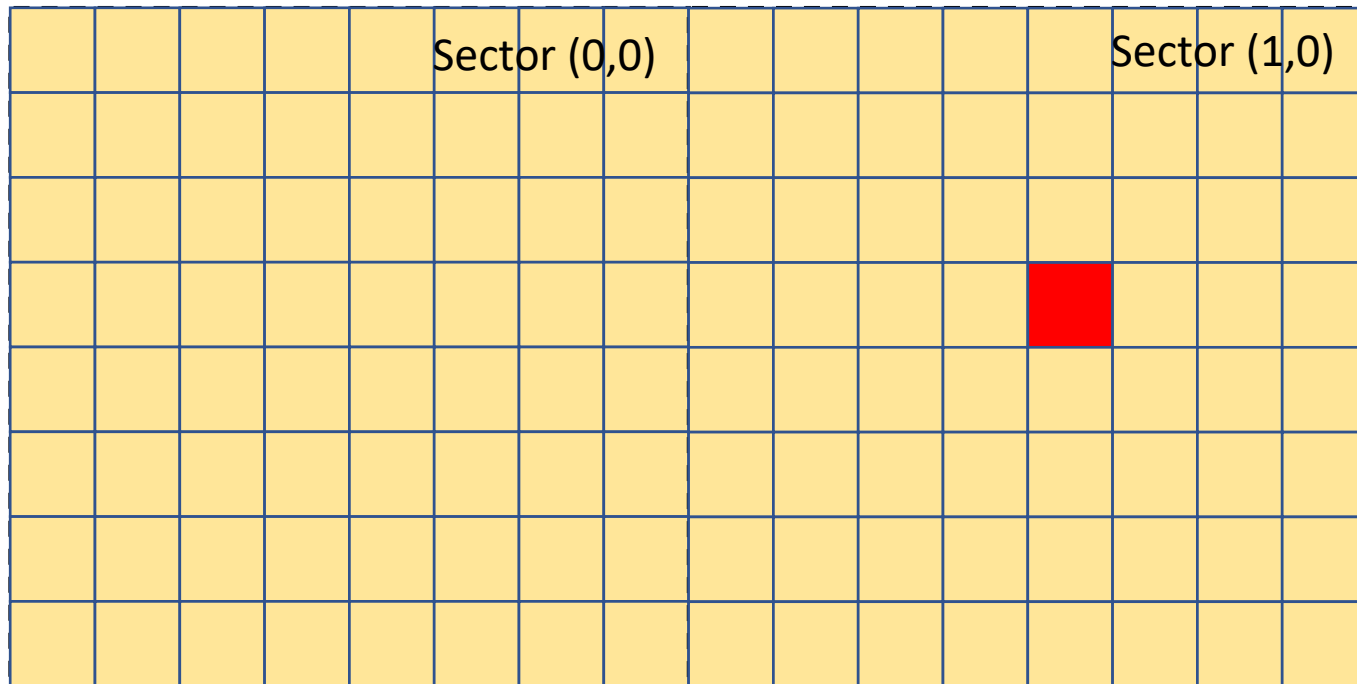


(0,0)에 액세스할때
ptr = AcquirePtr(0,0)
-> 8x8 메모리 할당
-> return sector (0,0)의 base
ptr + 0



(0,0)에 액세스할때
ptr = AcquirePtr(0,0)
-> 8x8 메모리 할당
-> return sector (0,0)의 base
ptr + 0

(8,1)에 액세스할때
ptr = AcquirePtr(8,1)
-> 8x8 메모리 할당
-> return sector (1,0)의 base
ptr + 8



Sector (2,0)

(0,0)에 액세스할때
ptr = AcquirePtr(0,0)
-> 8x8 메모리 할당
-> return sector (0,0)의 base
ptr + 0

(8,1)에 액세스할때
ptr = AcquirePtr(8,1)
-> 8x8 메모리 할당
-> return sector (1,0)의 base
ptr + 8

Sector (2,1)

(12,3)에 액세스할때
ptr = AcquirePtr(8,1)
-> 8x8 메모리 할당
-> return sector (1,0)의 base
ptr + 8

일반적인 tip

작업용 임시 메모리

- Stack(로컬 변수)는 대체로 cache hit한다. 따라서 작업용 메모리는 stack을 사용하는 편이 성능상 유리하다.
- 그러나 stack메모리를 너무 크게 잡으면 cache miss할 가능성이 높고 stack메모리를 추가 commit하느라 오히려 느려진다.
- 너무 큰 stack 변수(배열포함)를 사용하면 VC++에서 경고를 해준다.
- 큰 사이즈의 working 메모리가 필요할 경우 heap에다 잡아두고 사용한다.

멀티스레드 상황

- 스레드가 사용할 working 메모리는 한번에 할당해서 배정하는 것이 가장 좋다.
- 메모리 풀을 만들어 사용할 경우 lock이 필요없도록 아예 별도의 메모리 풀을 만들어서 배정한다.
- 스레드마다 heap을 사용할 필요가 있을 경우 별도의 heap을 만들어서 배정한다.

부지불식간에 사용중인 동적할당 제거

- STL 등 자료구조에서 new/delete를 사용할 경우 빈번하게 호출되고 있지 않은지 확인할 것.
- 메모리 풀 만들때 다른 자료구조 갖다 쓰려고 한다면 각별히 주의할 것. 그 자료구조에서 메모리 할당/해제로 CPU자원을 낭비할 수 있다.
- 함수 안에서 String 클래스 등 사용 주의.
- 렌더링 프레임에 malloc/free/new/delete 호출이 없도록 한다.
- 로딩이 느리다면 I/O비용을 의심하기 전에 메모리 할당/해제 비용부터 확인할 것.