



C++ STL

2019. 10. 01

정재훈



목차

- List
 - Memory Alignment
- Vector
- Set
- Map
- Unordered Map



앞으로 설명에서 계속 쓰일 것들

```
struct Obj{  
    char a[16];  
};
```

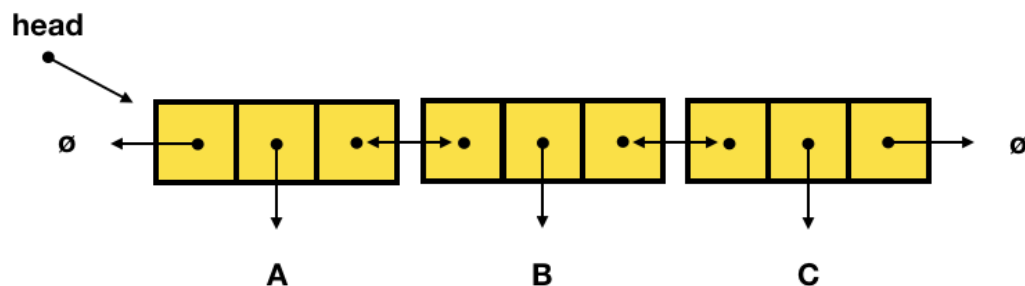
gmem : malloc(new)이나 free(delete) 될 때 할당 된 메모리 크기와 주소를 나타내 줌

List

C++ 에서는 Double linked list로 구현되어 있음

```
#include <list>
```

```
void testList() {  
    list<Obj> l;  
    Obj obj1, obj2, obj3, obj4;  
  
    l.push_back(obj1);  
    l.push_back(obj2);  
    l.push_back(obj3);  
    l.push_back(obj4);  
}
```



List

```
28
29 void testList(){
30     list<Obj> l;
31     Obj obj1, obj2, obj3, obj4;
32
33     l.push_back(obj1);
34     l.push_back(obj2);
35     l.push_back(obj3);
36     l.push_back(obj4);
37 }
38
```

Terminal

File Edit View Search Terminal Help

[gmem] new(32) > 0x55f276713e70

v.reserve(5); //80

v.push_back(obj1); //16

v.push_back(obj1); //32

v.push_back(obj1); //64

Debugger GDB for "stl_test"

Level Function

1 testList

2 main

List

```
29 void testList(){
30     list<Obj> l;
31     Obj obj1, obj2, obj3, obj4;
32
33     l.push_back(obj1);
34     l.push_back(obj2);
35     l.push_back(obj3);
36     l.push_back(obj4);
37 }
```

Terminal

File Edit View Search Terminal Help

4 [gmem] new(32) > 0x55f276713e70

4 [gmem] new(32) > 0x55f276714300

4

43 v.reserve(5); //80

44 v.push_back(obj1); //16

45 v.push_back(obj1); //32

Debugger GDB for "stl_test"

List

```
29 void testList(){  
30     list<Obj> l;  
31     Obj obj1, obj2, obj3, obj4;  
32  
33     l.push_back(obj1);  
34     l.push_back(obj2);  
35     l.push_back(obj3);  
36     l.push_back(obj4);  
37 }  
38  
39  
40  
41  
42  
43  
44  
45  
46  
47
```

Terminal

File Edit View Search Terminal Help

[gmem] new(32) > 0x55631359ee70

[gmem] new(32) > 0x55631359f300

[gmem] new(32) > 0x55631359f360

[gmem] new(32) > 0x55631359f400

[gmem] delete(0x55631359ee70)

[gmem] delete(0x55631359f300)

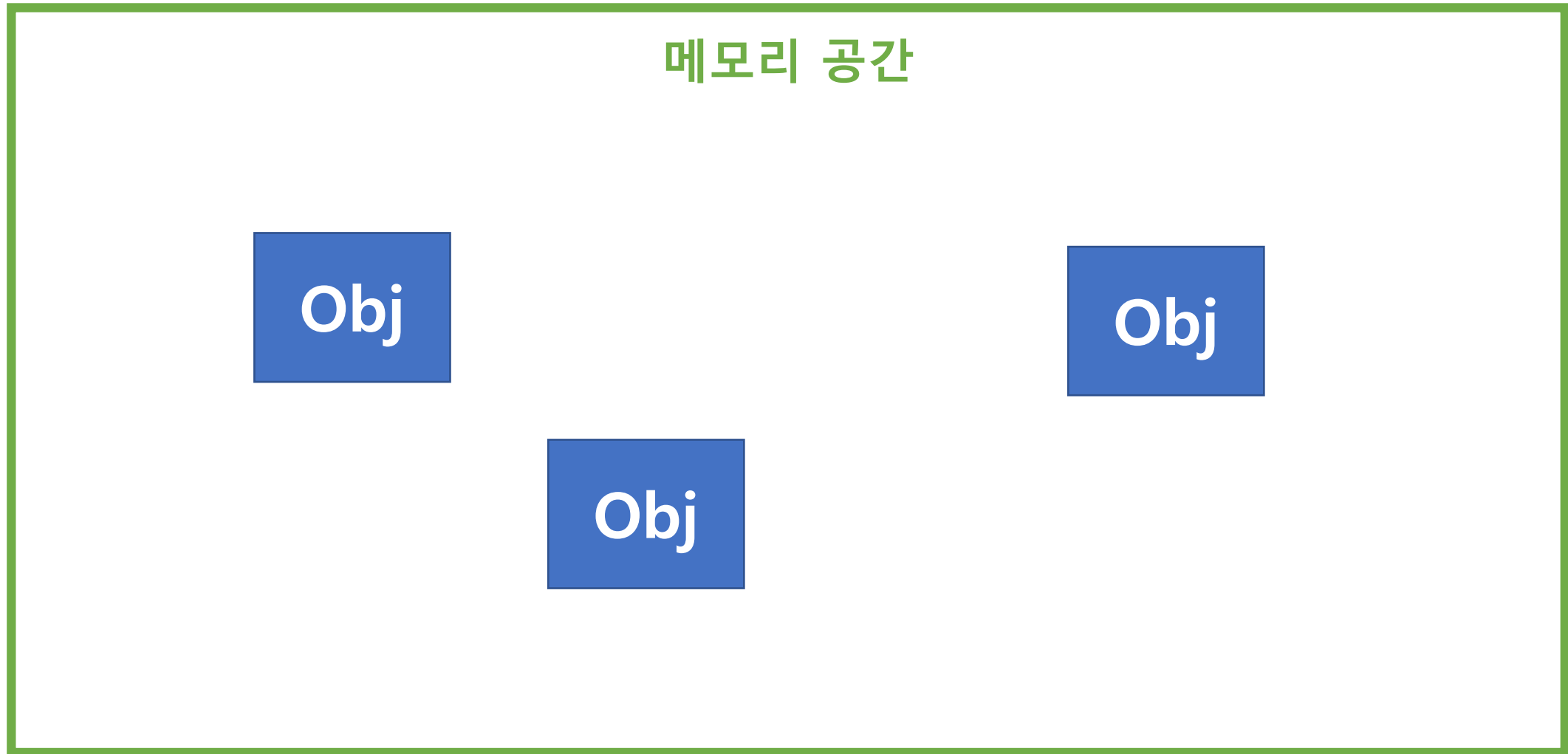
[gmem] delete(0x55631359f360)

[gmem] delete(0x55631359f400)

Debugger

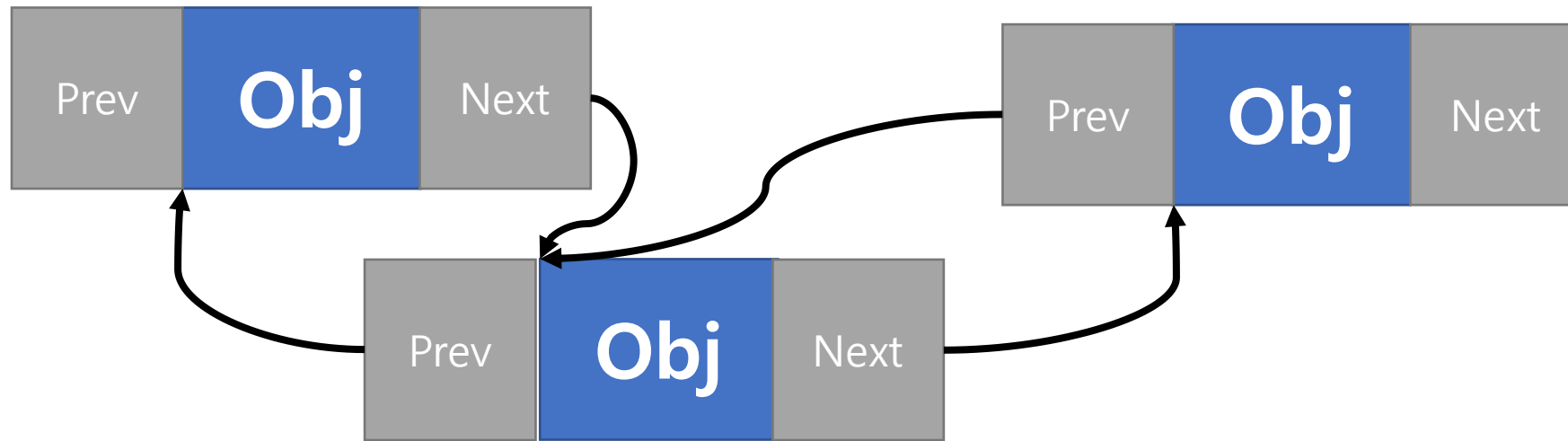
GDB for "sh: test" Threads:

List



List

메모리 공간



List

```
16 void testInt(){
17     list<int> l;
18
19     l.push_back(1);
20     l.push_back(2);
21     l.push_back(3);
22     l.push_back(4);
23
24     list<int>::iterator it;
25     for(it = l.begin(); it != l.end(); it++)
26         printf("%d\n", *it);
27 }
```

Terminal

```
File Edit View Search Terminal Help
[gmem] new(24) > 0x5625aa132e70, obj4;
[gmem] new(24) > 0x5625aa1332f0
[gmem] new(24) > 0x5625aa133340
[gmem] new(24) > 0x5625aa132ec0
```

Debugger GDB for "stl_test" Threads: #1 stl_t

List

```
16 void testInt(){
17     list<char> l;
18
19     l.push_back(1);
20     l.push_back(2);
21     l.push_back(3);
22     l.push_back(4);
23
24     list<char>::iterator it;
25     for(it = l.begin(); it != l.end(); it++)
26         printf("%d\n", *it);
27 }
```

Terminal

File Edit View Search Terminal Help

```
[gmem] new(24) > 0x556fffa16e70
[gmem] new(24) > 0x556fffa172f0
[gmem] new(24) > 0x556fffa17340
[gmem] new(24) > 0x556fffa16ec0
```

Debugger: GDB for "stl_test" Threads: #1

List



8bytes

1byte

8bytes

17bytes????????????

List



32 bit →
64 bit →

4bytes
8bytes

최소 4bytes
최소 8bytes

4bytes →
8bytes →

최소 12bytes
최소 24bytes

Memory Alignment

컴퓨터 메모리에서 데이터가 배열되고 액세스되는 방식을 나타냅니다.

데이터 정렬, 데이터 구조 패딩 및 패킹의 세 가지 개별적인 관련 문제로 구성됩니다.

최신 컴퓨터 하드웨어의 CPU는 데이터가 자연스럽게 정렬 될 때 가장 효율적으로 메모리에 읽고 쓰기를 수행합니다.

Vector

쉽게 말해 '가변 배열'임
할당한 만큼보다 배열에 추가 될 경우 재할당이 일어 남(2배 씩)

```
#include <vector>

void testVector(){
    vector<Obj> v;
    Obj obj1;

    v.push_back(obj1);
}
```

Vector

```
void testVector(){  
    vector<Obj> v;  
    Obj obj1;
```

```
    v.push_back(obj1); //16
```

```
    v.push_back(obj1); //32
```

```
    v.push_back(obj1); //64
```

```
    v.push_back(obj1);
```

```
}
```

Terminal

File Edit View Search Terminal Help

[gmem] new(16) > 0x55e024180e70

[gmem] new(32) > 0x55e0241812f0

[gmem] delete(0x55e024180e70)

[gmem] new(64) > 0x55e024181350

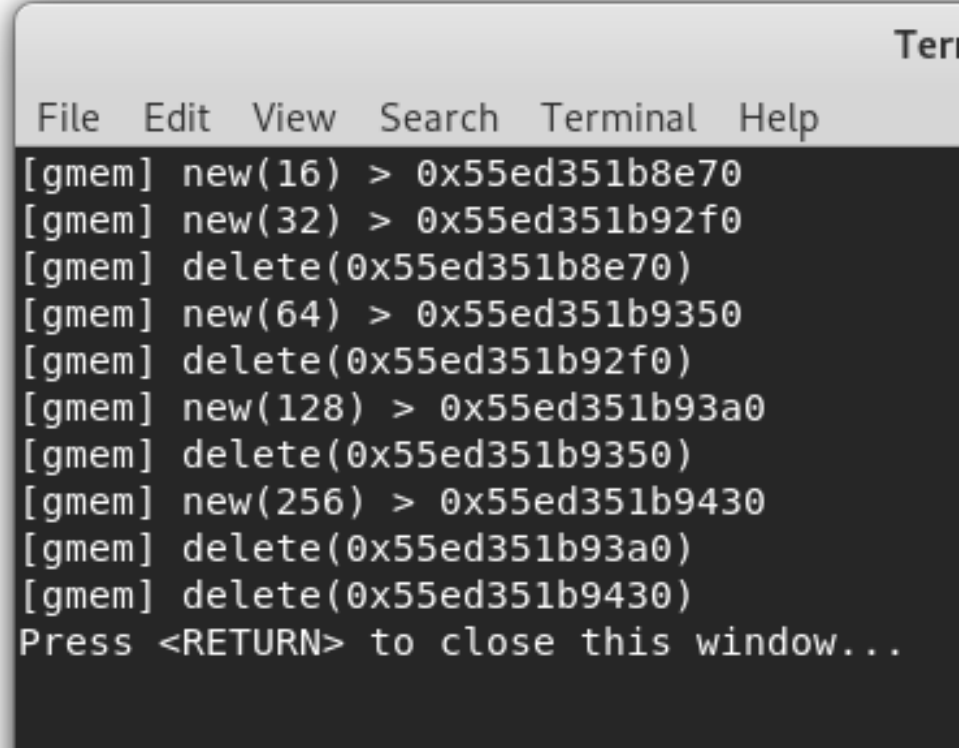
[gmem] delete(0x55e0241812f0)

[gmem] delete(0x55e024181350)

Press <RETURN> to close this window...

Vector

```
void testVector(){  
    vector<Obj> v;  
    Obj obj1;  
  
    v.push_back(obj1); //16  
    v.push_back(obj1); //32  
    v.push_back(obj1); //64  
    v.push_back(obj1);  
    v.push_back(obj1); //128  
    v.push_back(obj1);  
    v.push_back(obj1);  
    v.push_back(obj1);  
    v.push_back(obj1); //256  
}
```



The screenshot shows a terminal window titled "Terminal" with a menu bar containing "File", "Edit", "View", "Search", "Terminal", and "Help". The terminal output displays the memory management of a vector as it grows. It shows the allocation of memory blocks of sizes 16, 32, 64, 128, and 256 bytes, each with a corresponding hexadecimal address. After each allocation, the previous block is deallocated. The final state shows a 256-byte block at address 0x55ed351b9430. The prompt "Press <RETURN> to close this window..." is visible at the bottom.

```
Terminal  
File Edit View Search Terminal Help  
[gmem] new(16) > 0x55ed351b8e70  
[gmem] new(32) > 0x55ed351b92f0  
[gmem] delete(0x55ed351b8e70)  
[gmem] new(64) > 0x55ed351b9350  
[gmem] delete(0x55ed351b92f0)  
[gmem] new(128) > 0x55ed351b93a0  
[gmem] delete(0x55ed351b9350)  
[gmem] new(256) > 0x55ed351b9430  
[gmem] delete(0x55ed351b93a0)  
[gmem] delete(0x55ed351b9430)  
Press <RETURN> to close this window...
```

Vector

메모리 공간

Obj

Vector

메모리 공간



Vector

메모리 공간



Vector

메모리 공간



Vector

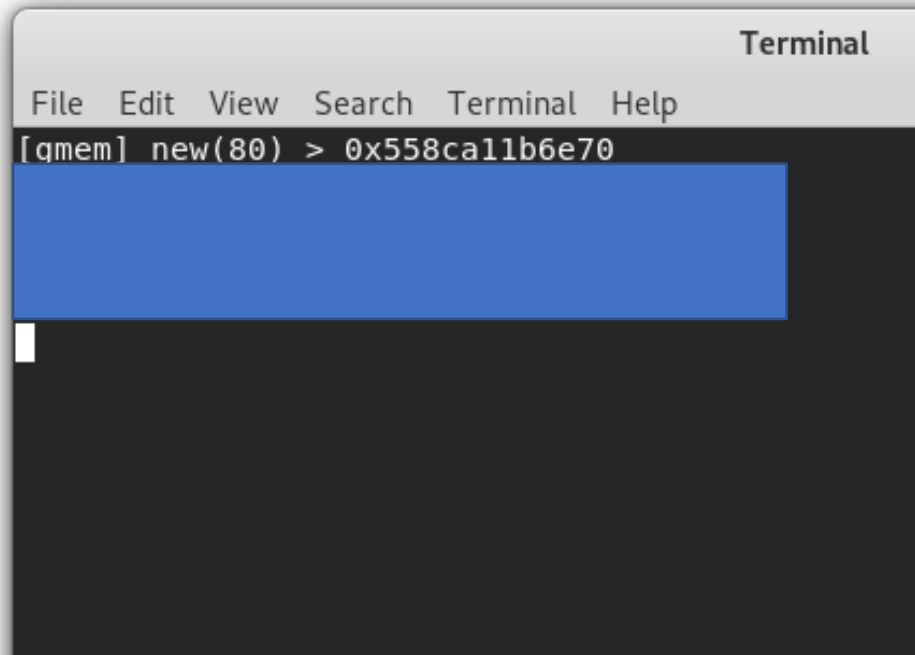
메모리 공간



Vector

처음 생성 할 공간을 지정해 줄 수 있다.

```
void testVector(){  
    vector<Obj> v;  
    Obj obj1;  
  
    v.reserve(5);    //80  
    v.push_back(obj1); //16  
    v.push_back(obj1); //32  
    v.push_back(obj1); //64  
    v.push_back(obj1);  
    v.push_back(obj1); //128  
    v.push_back(obj1);  
    v.push_back(obj1);  
    v.push_back(obj1);  
    v.push_back(obj1); //256  
}
```



List와 Vector

list

- 중간 삽입/삭제 가능
- 순차접근 가능
- 많은 양의 자료에 불리
- 랜덤 액세스 불가(오프셋으로 접근 X)
- 검색 느림(순차 접근 때문 – 무조건 $O(n)$)

vector

- 중간 삽입/삭제 불가
- 순차접근 가능
- 많은 양의 자료에 불리
- 랜덤 액세스 가능
- 검색 느림(리스트와 같은 이유)

* 중간 삽입/삭제가 없고 랜덤접근이 많다 : vector

* 중간 삽입/삭제가 있고 랜덤접근이 없다 : list

* 하지만 데이터 개수가 적은 경우는 vector 사용하는 편이 더 좋다(성능상)

- gMem : <https://github.com/snoopsy/gmem>
- vector - <https://hyeonstorage.tistory.com/324>
- list - <https://hyeonstorage.tistory.com/326>