Project Blizzard
Safe manual memory management

Alexandru Jercaianu
University POLITEHNICA of Bucharest
alex.jercaianu@gmail.com

DConf 2018
Munich, May 2-5, 2018

Manual memory management

High performance

Deterministic lifetimes

Garbage collector

Safety

Increase in productivity

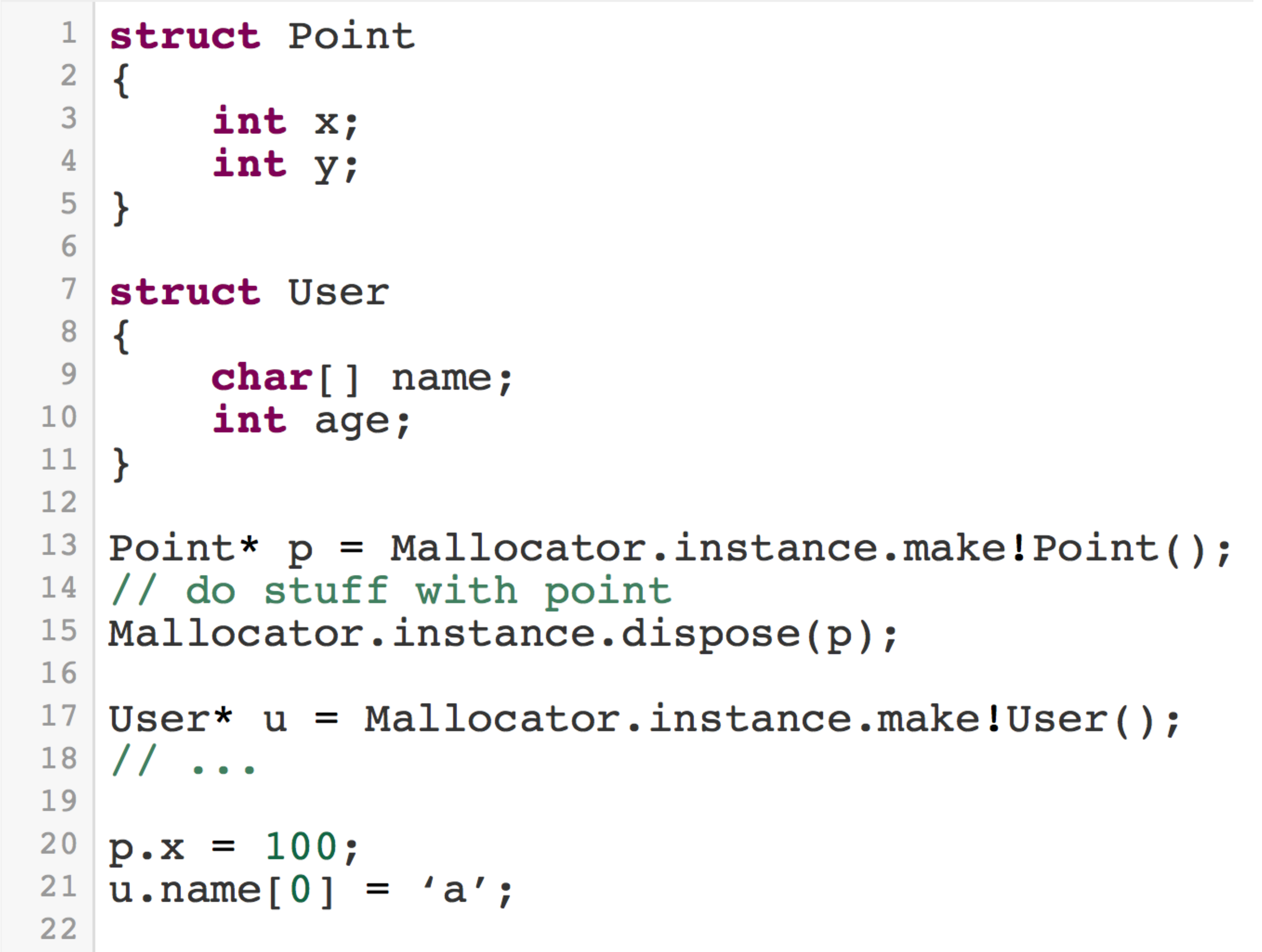
Vulnerabilities

Use after free

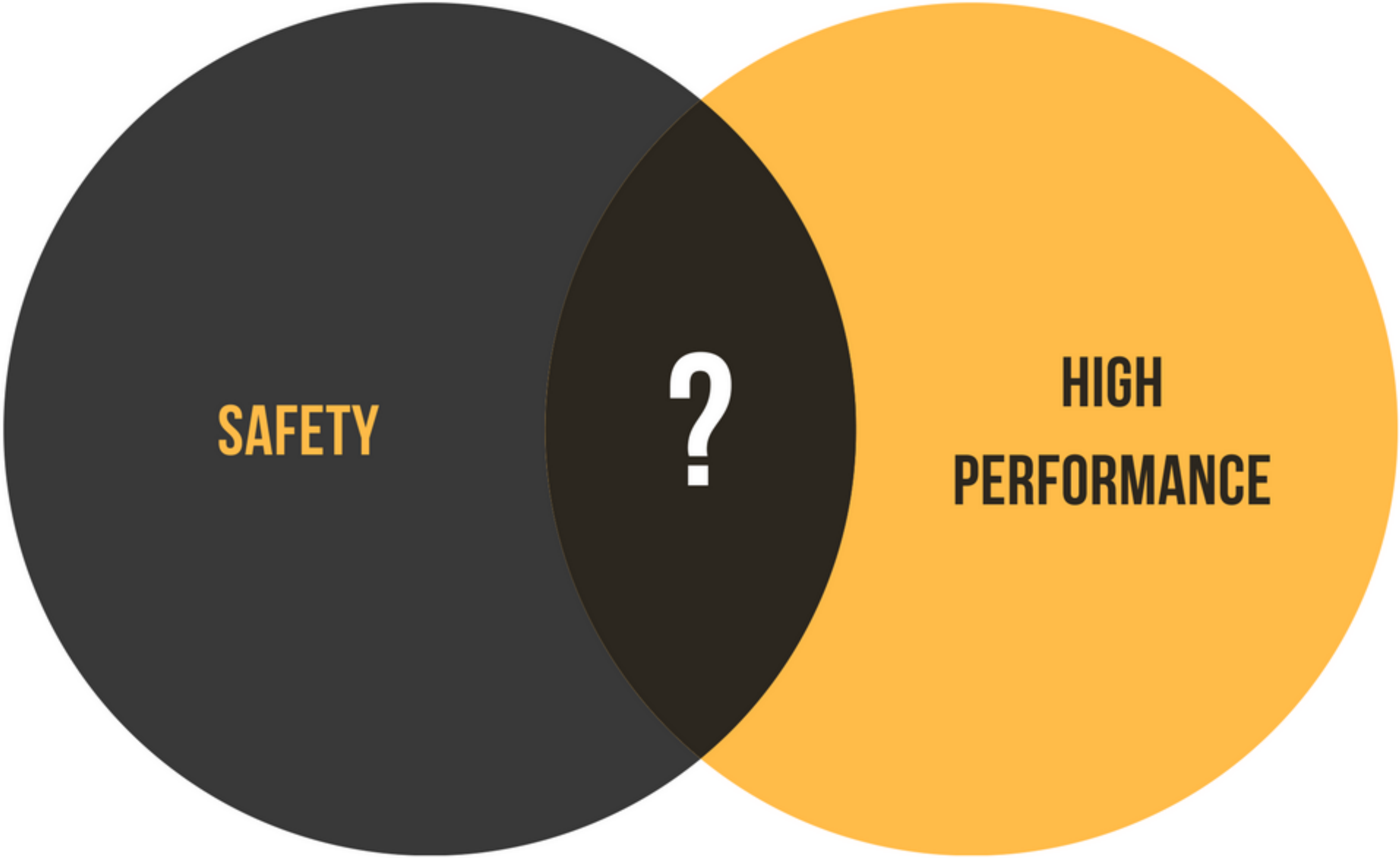
Buffer overflows

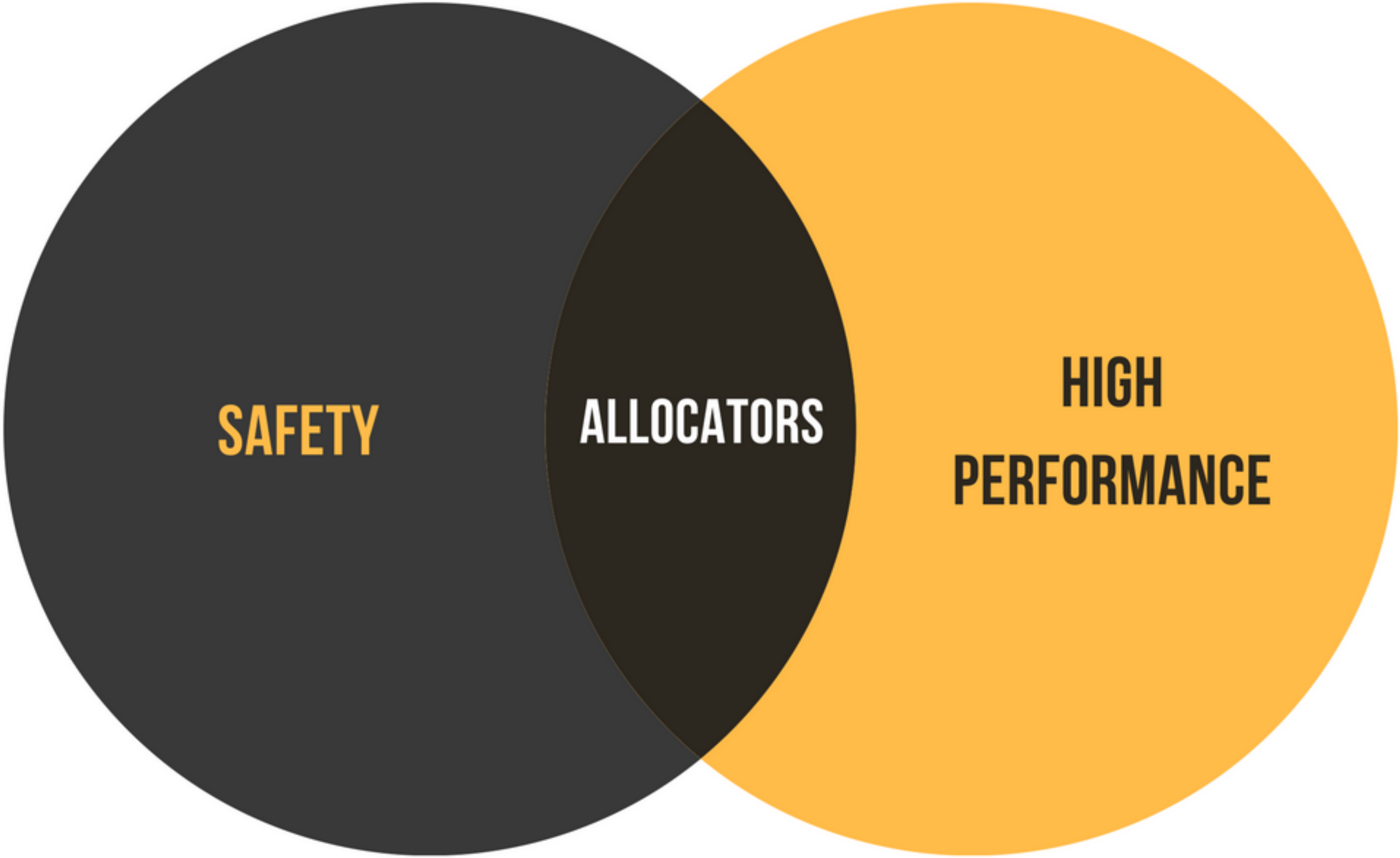
Double free

Undefined behavior



```
struct Point
{
    int x;
    int y;
}

struct User
{
    char[] name;
    int age;
}

Point* p = Mallocator.instance.make!Point();
// do stuff with point
Mallocator.instance.dispose(p);

User* u = Mallocator.instance.make!User();
// ...

p.x = 100;
u.name[0] = ‘a’;

```


SAFETY
?
HIGH
PERFORMANCE



SAFETY

ALLOCATORS

HIGH PERFORMANCE

Blizzard allocator

Usable in safe code

Mitigate dangling pointers

High performance

Coexist with other allocators

Related work

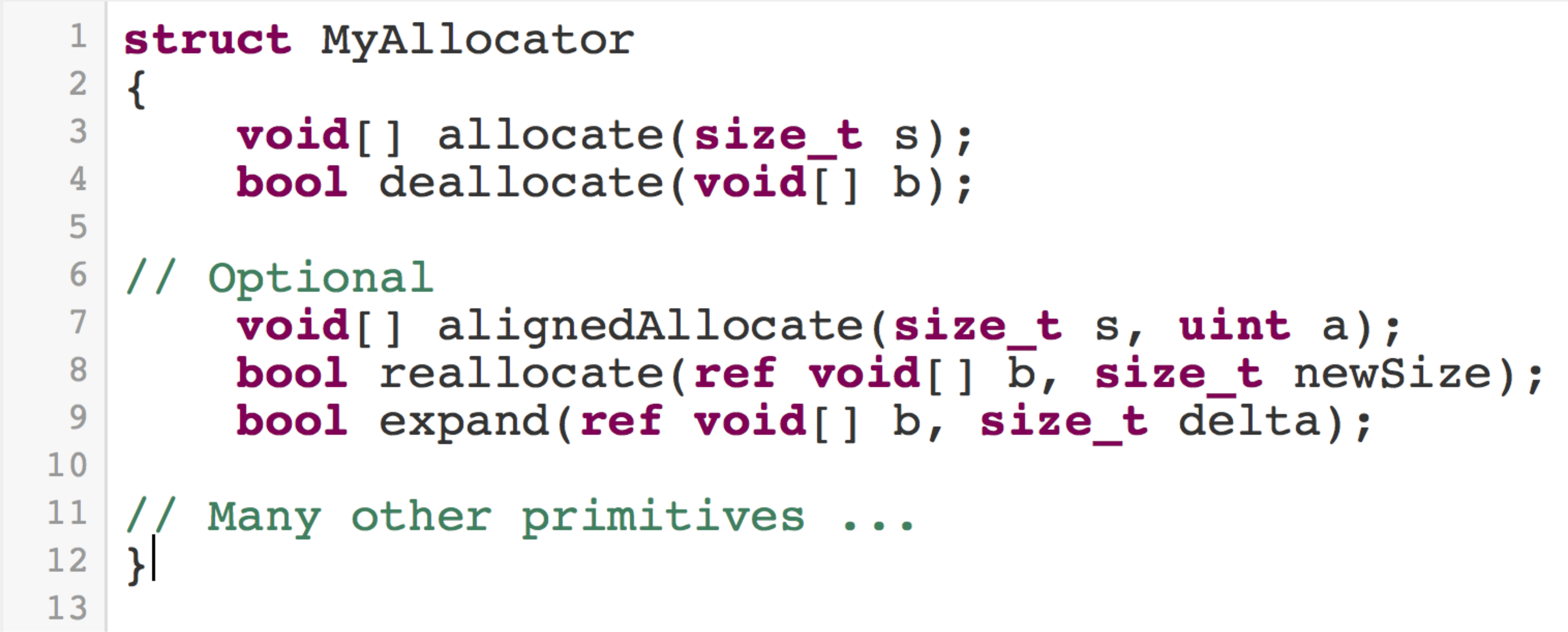
- Project Snowflake: Non-blocking safe manual memory management in .NET

(https://www.microsoft.com/en-us/research/publication/project-snowflake-non-blocking-safe-manual-memory-management-net/)

- Simple, Fast and Safe Manual Memory Management

(https://www.microsoft.com/en-us/research/publication/simple-fast-safe-manual-memory-management/)

Allocators



```
struct MyAllocator
{
    void[] allocate(size_t s);
    bool deallocate(void[] b);

// Optional
    void[] alignedAllocate(size_t s, uint a);
    bool reallocate(ref void[] b, size_t newSize);
    bool expand(ref void[] b, size_t delta);

// Many other primitives ...
}
```

Allocator examples

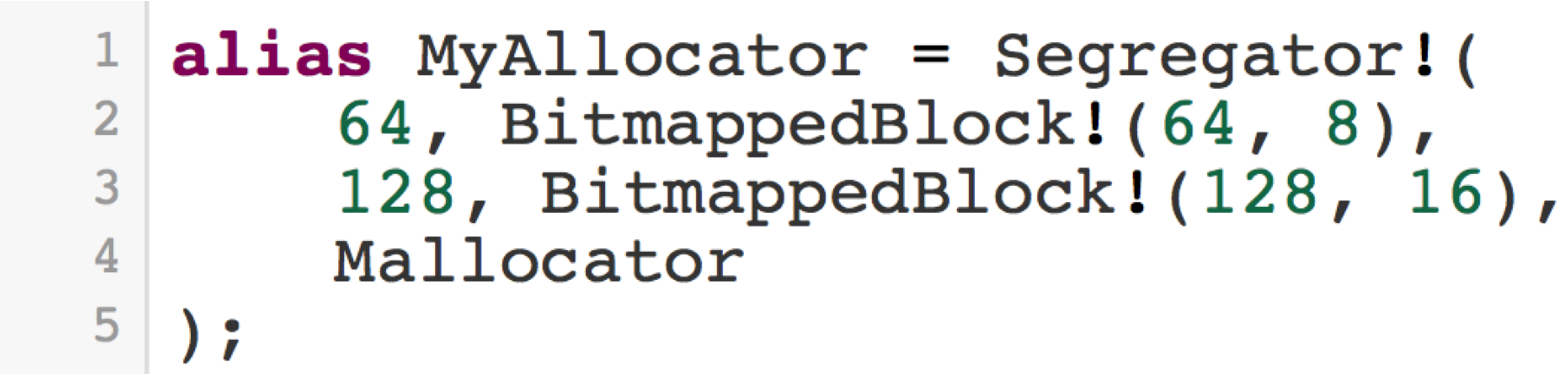
Region

BitmappedBlock

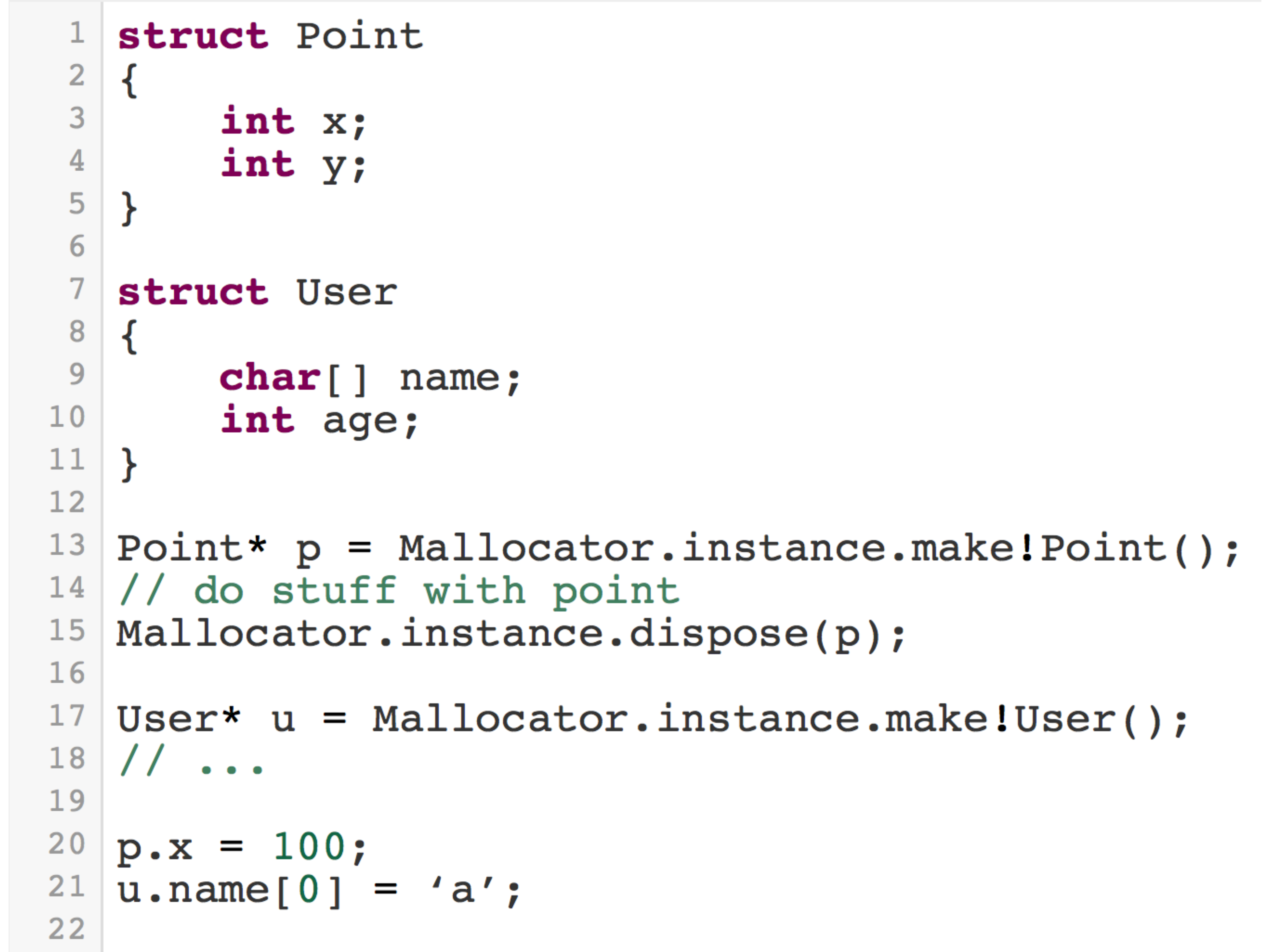
GCAllocator

Mallocator

Building blocks



```
alias MyAllocator = Segregator!(
    64, BitmappedBlock!(64, 8),
    128, BitmappedBlock!(128, 16),
    Mallocator
);
```



```
struct Point
{
      int x;
      int y;
}

struct User
{
      char[] name;
      int age;
}

Point* p = Mallocator.instance.make!Point();
// do stuff with point
Mallocator.instance.dispose(p);

User* u = Mallocator.instance.make!User();
// ...

p.x = 100;
u.name[0] = ‘a’;

```

Problem

Reusing memory is not safe

Problem

Reusing memory is not safe

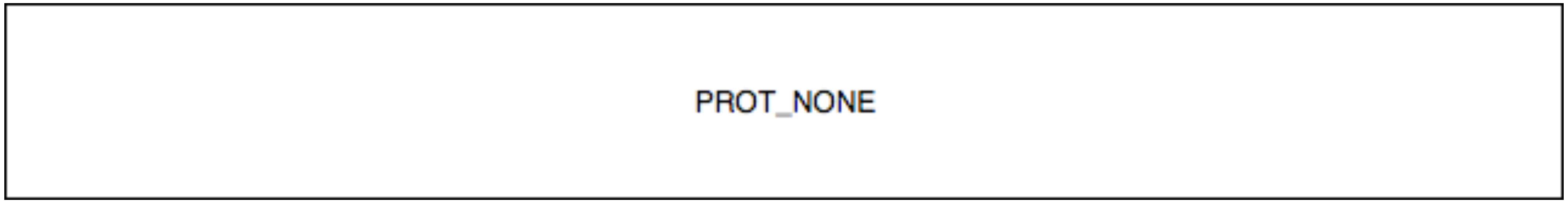
Solution

Always allocate at increasing addresses

Great for 64bit

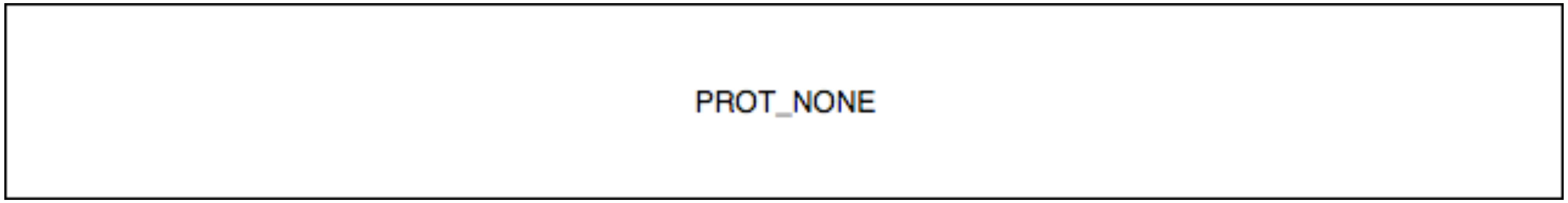
Ascending Page Allocator

Map a large chunk of virtual memory with no permissions

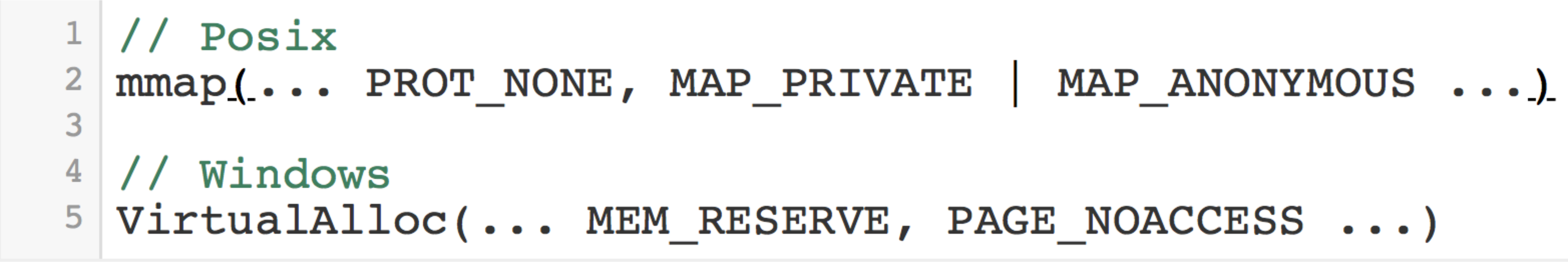


Ascending Page Allocator

Map a large chunk of virtual memory with no permissions



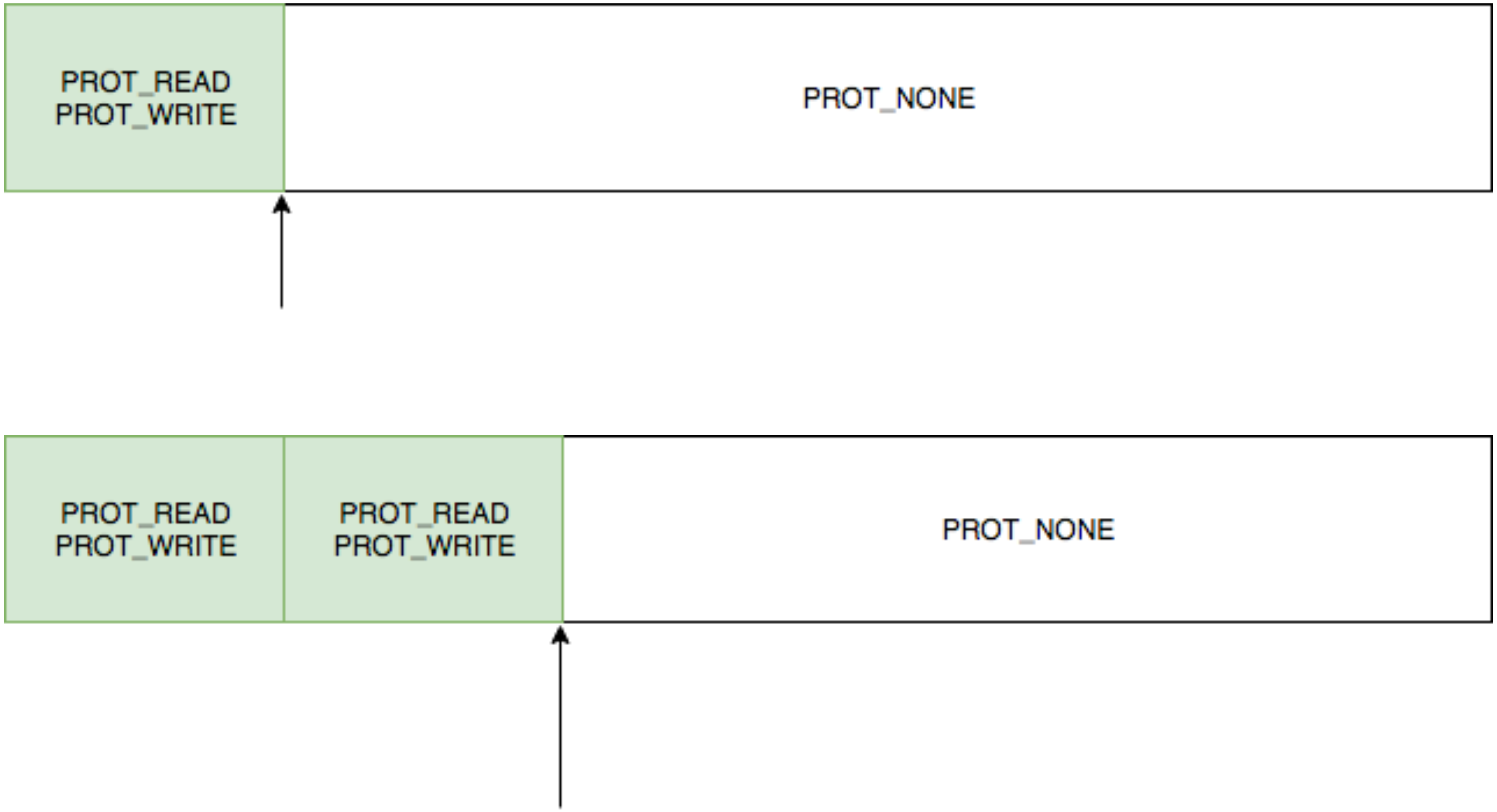
PROT_NONE



```
// Posix
mmap(... PROT_NONE, MAP_PRIVATE | MAP_ANONYMOUS ...)

// Windows
VirtualAlloc(... MEM_RESERVE, PAGE_NOACCESS ...)
```

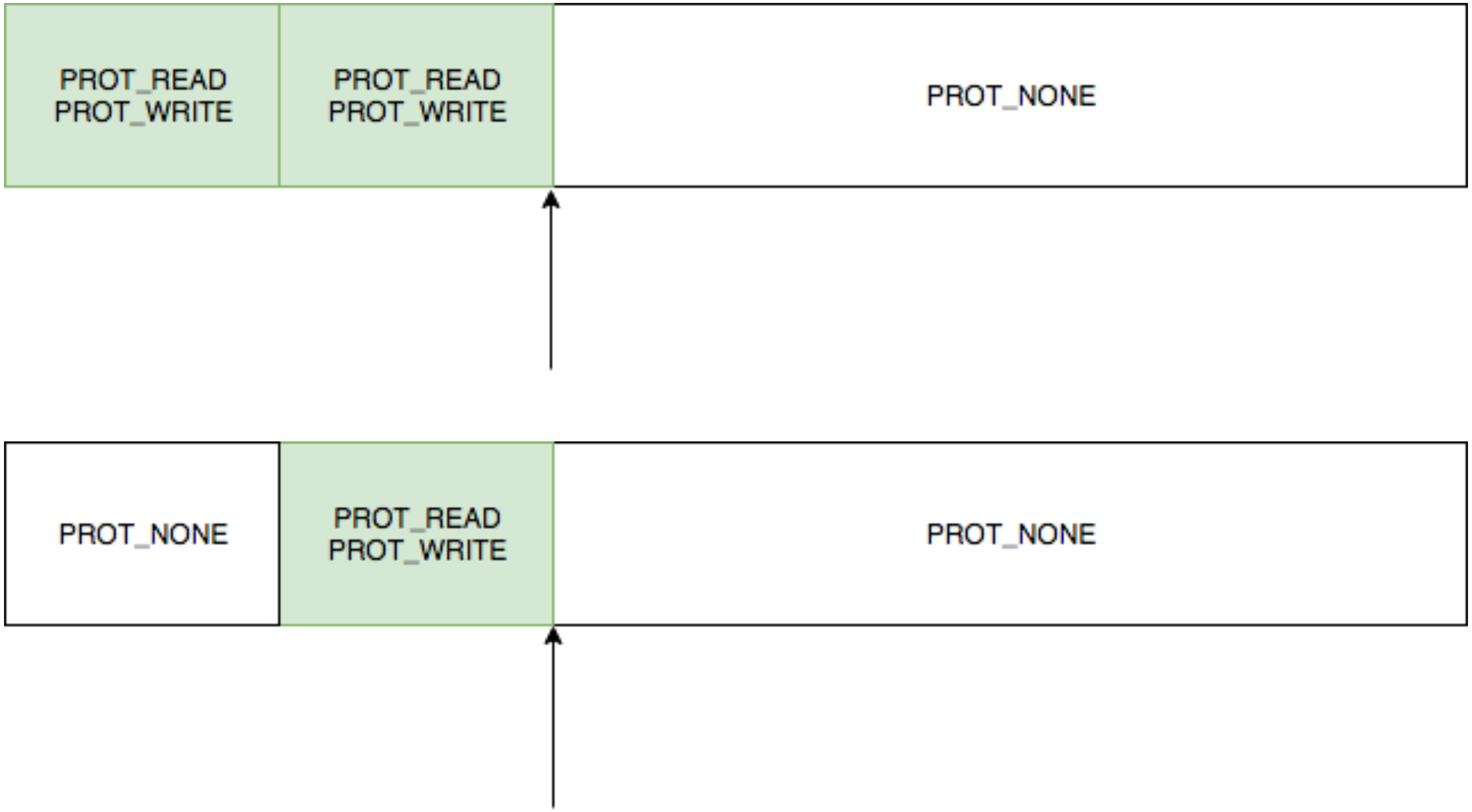
Ascending Page Allocator

Each allocation advances a pointer and sets read/write permissions



Ascending Page Allocator

Reclaim physical pages
Remove read/write permissions



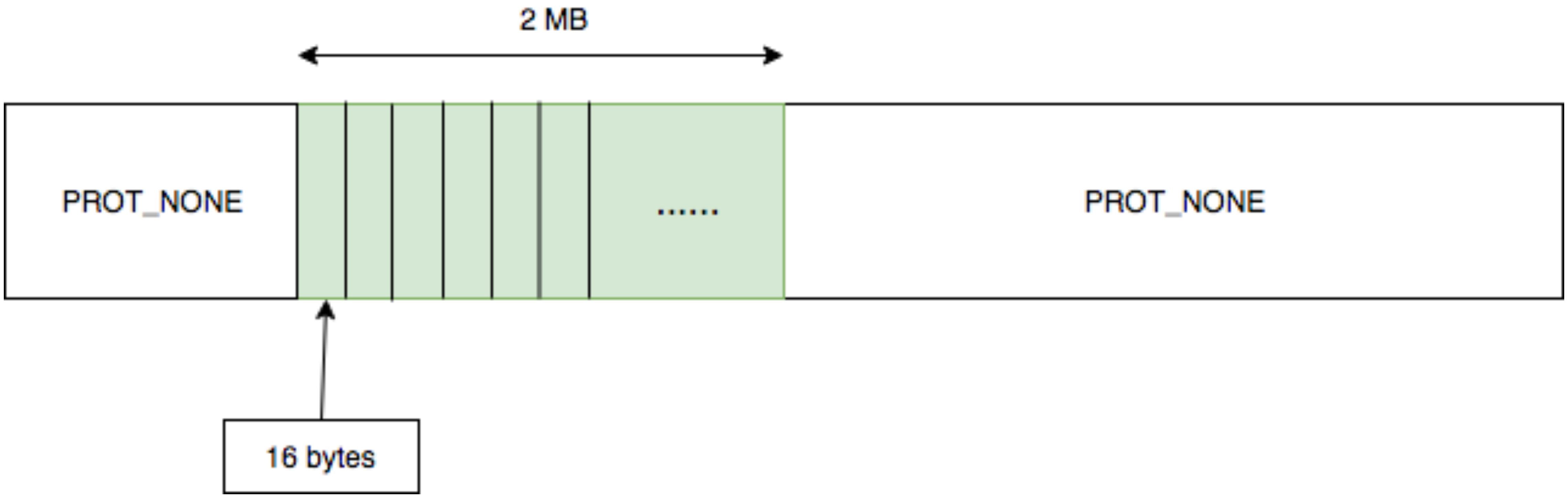
Ascending Page Allocator

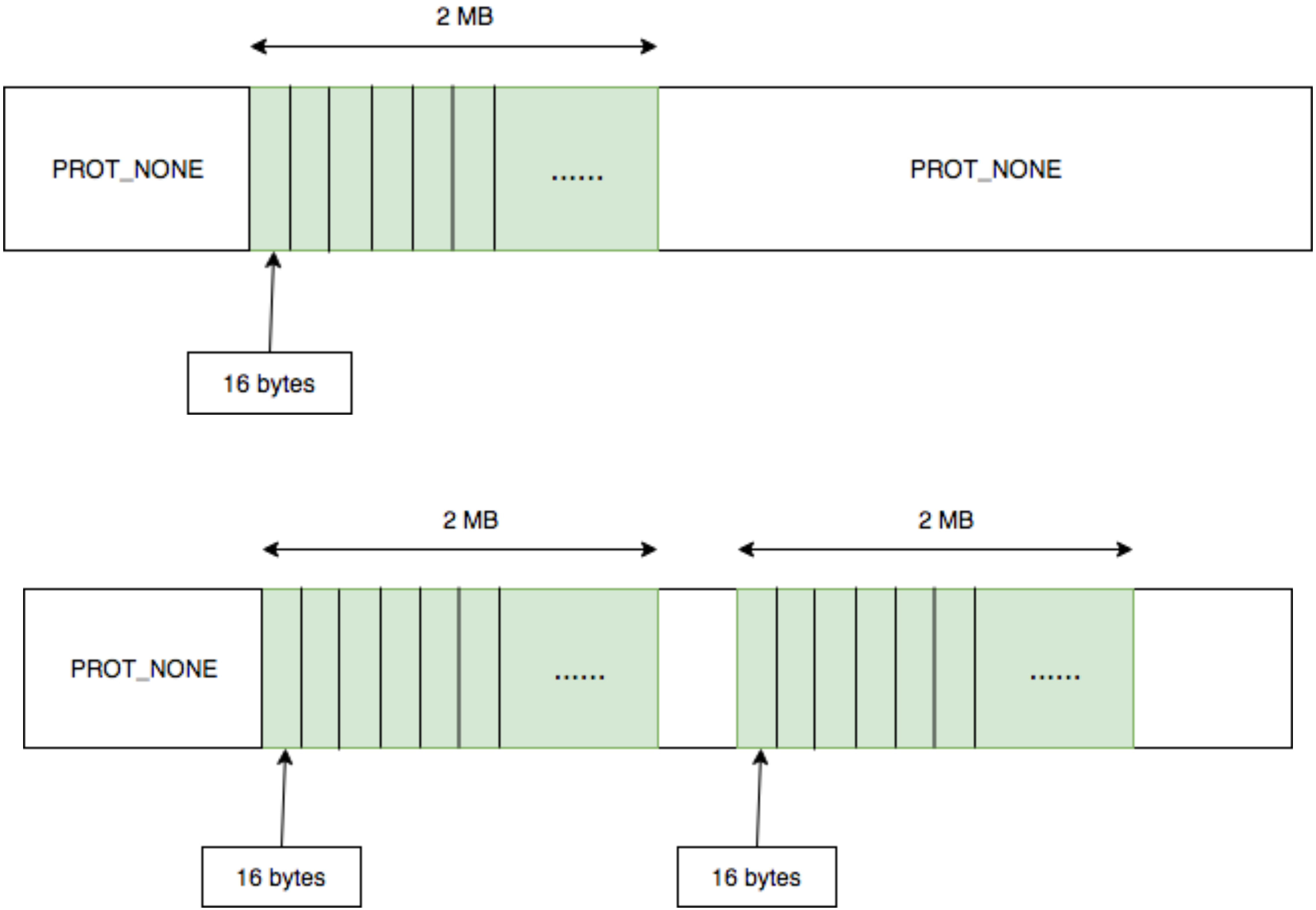
Safe

Page granularity

Great for huge allocations

High fragmentation


2 MB
PROT_NONE
......
PROT_NONE
16 bytes

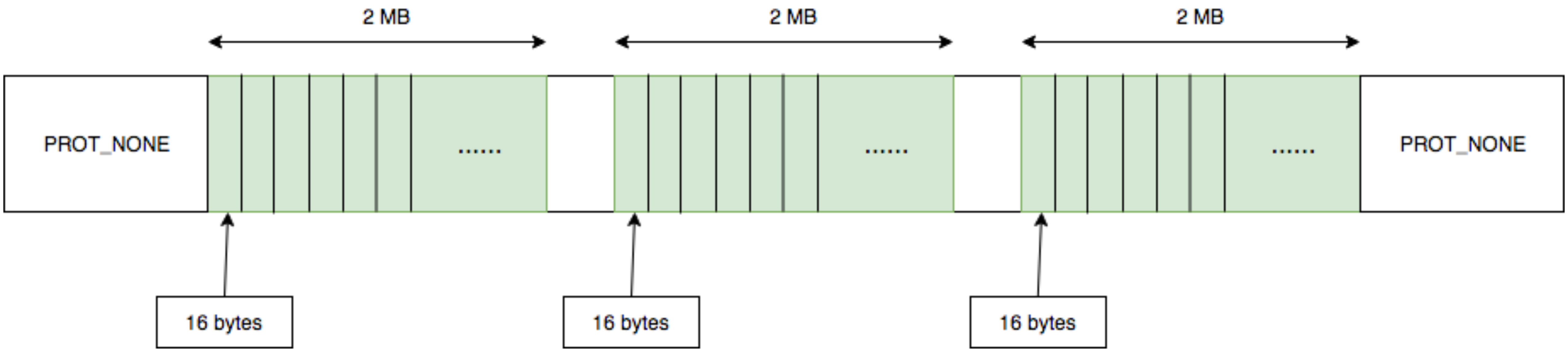

2 MB
PROT_NONE
......
PROT_NONE
16 bytes
2 MB
2 MB
PROT_NONE
......
......
16 bytes
16 bytes

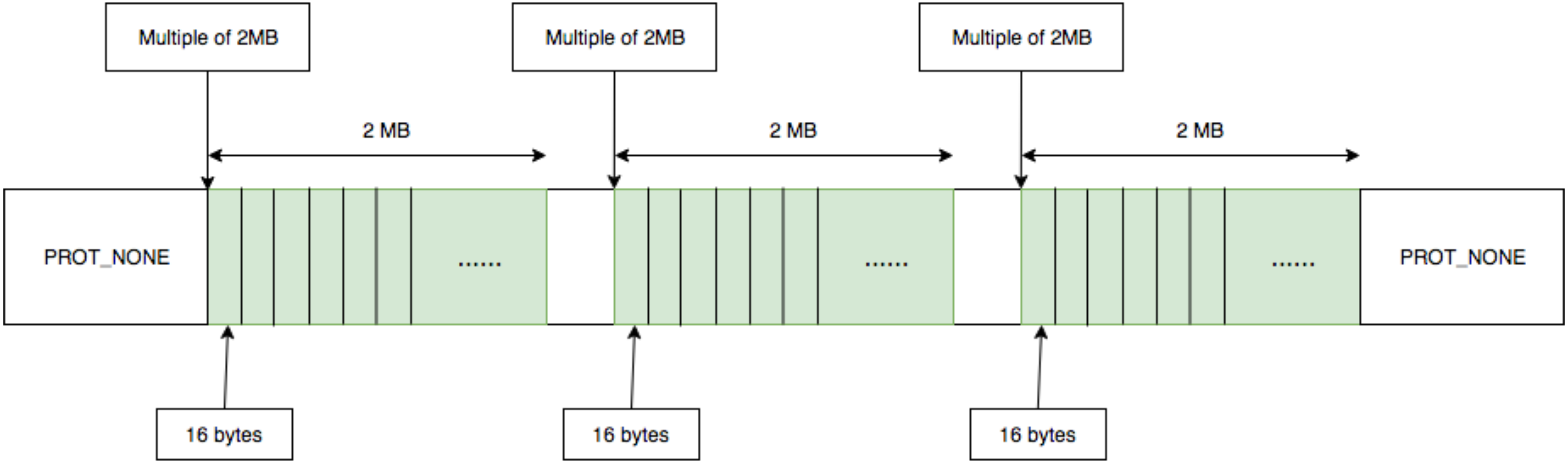
Allocation

- Return the first unused 16 byte block
- Remember, do not reuse memory

Deallocation

- Given a 16 byte block, how to quickly find the corresponding 2MB chunk?

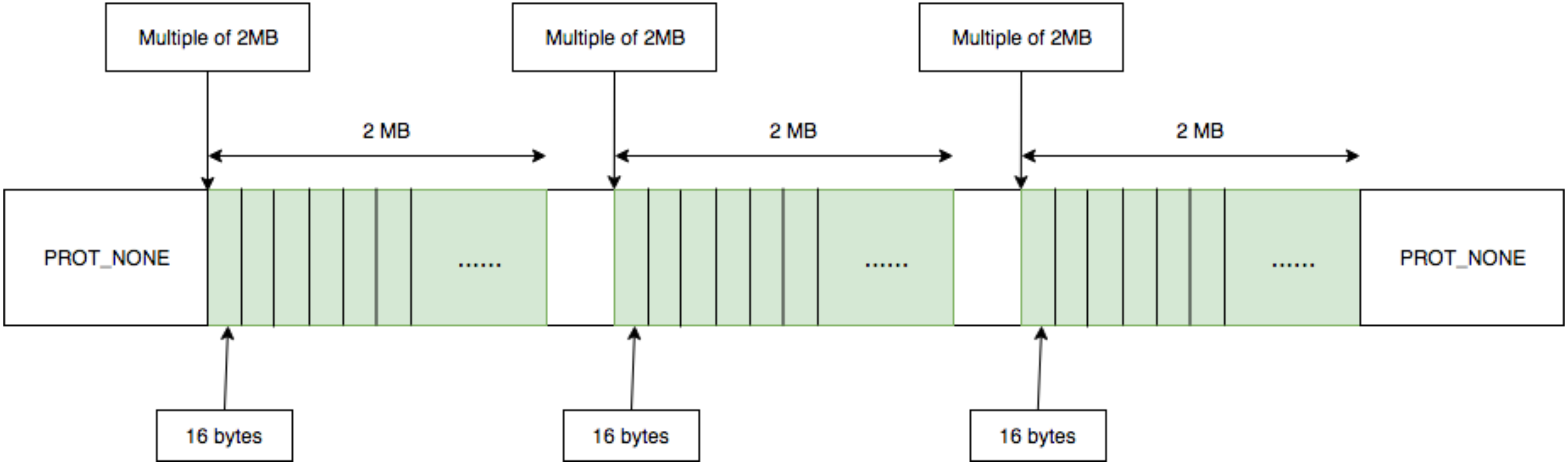

2 MB
2 MB
2 MB
PROT_NONE
......
......
......
PROT_NONE
16 bytes
16 bytes
16 bytes


Multiple of 2MB
Multiple of 2MB
Multiple of 2MB
2 MB
2 MB
2 MB
PROT_NONE
......
......
......
PROT_NONE
16 bytes
16 bytes
16 bytes

Aligned Block List

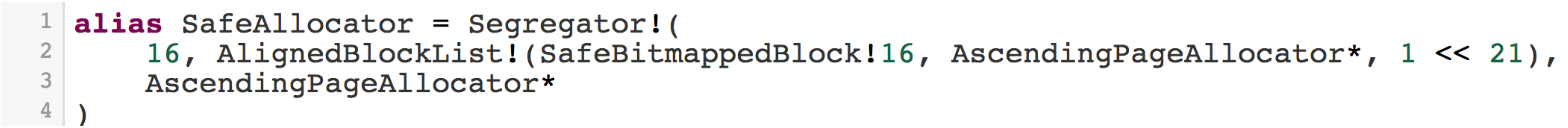
- Linked list of huge blocks, which are managed by another allocator (eg. BitmappedBlock)

- Length of each block is equal to its alignment, allowing for fast deallocations

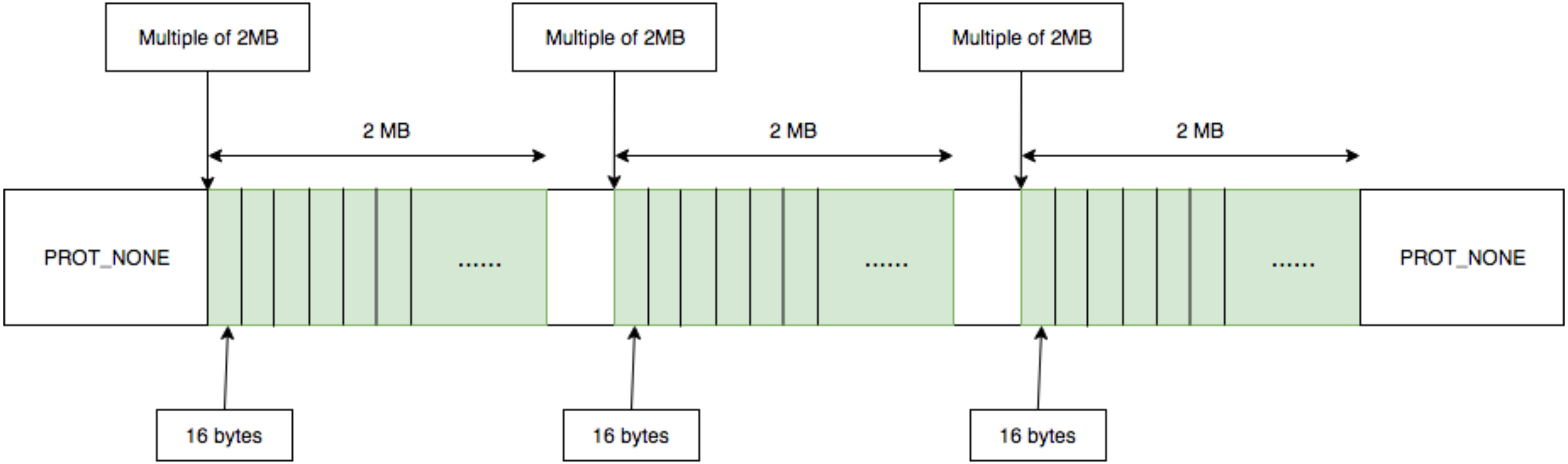


- SafeBitmappedBlock
- AlignedBlockList
- AscendingPageAllocator

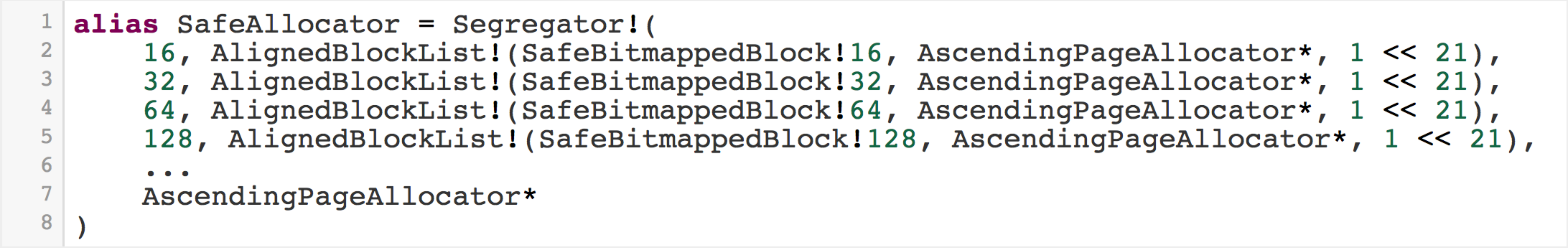
SafeAllocator v1.0



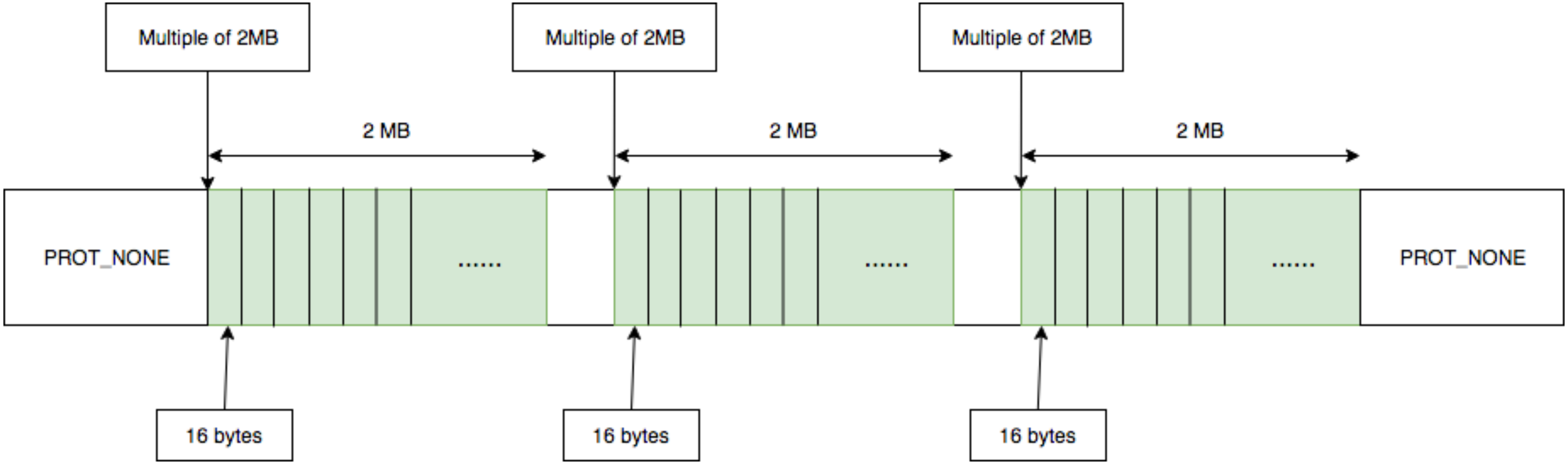
```
alias SafeAllocator = Segregator!(
    16, AlignedBlockList!(SafeBitmappedBlock!16, AscendingPageAllocator*, 1 << 21),
    AscendingPageAllocator*
)
```

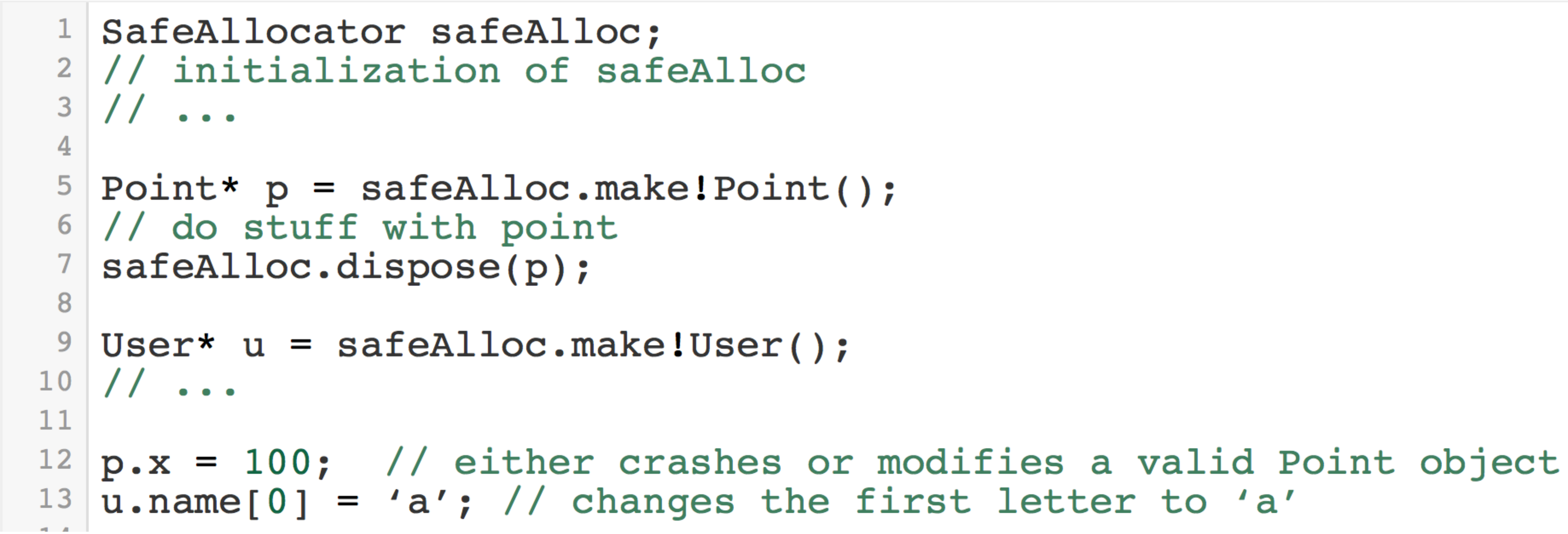


SafeAllocator v1.0



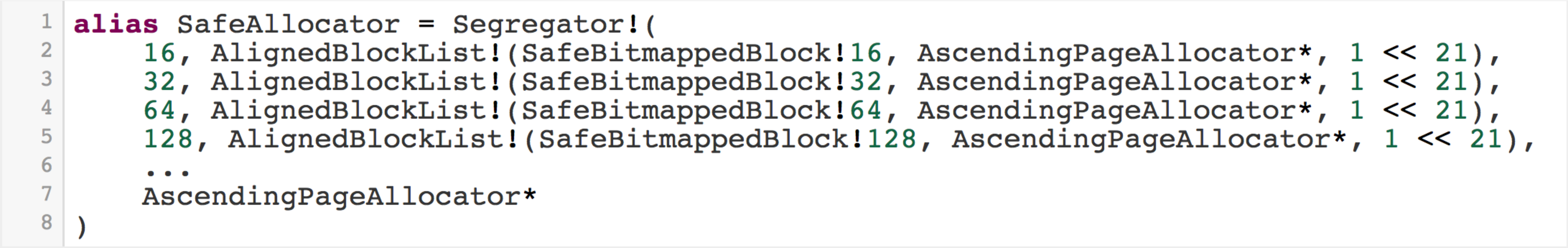
```d
alias SafeAllocator = Segregator!(
    16, AlignedBlockList!(SafeBitmappedBlock!16, AscendingPageAllocator*, 1 << 21),
    32, AlignedBlockList!(SafeBitmappedBlock!32, AscendingPageAllocator*, 1 << 21),
    64, AlignedBlockList!(SafeBitmappedBlock!64, AscendingPageAllocator*, 1 << 21),
    128, AlignedBlockList!(SafeBitmappedBlock!128, AscendingPageAllocator*, 1 << 21),
    ...
    AscendingPageAllocator*
)
```





```
SafeAllocator safeAlloc;
// initialization of safeAlloc
// ...

Point* p = safeAlloc.make!Point();
// do stuff with point
safeAlloc.dispose(p);

User* u = safeAlloc.make!User();
// ...

p.x = 100;  // either crashes or modifies a valid Point object
u.name[0] = ‘a’; // changes the first letter to ‘a’
```

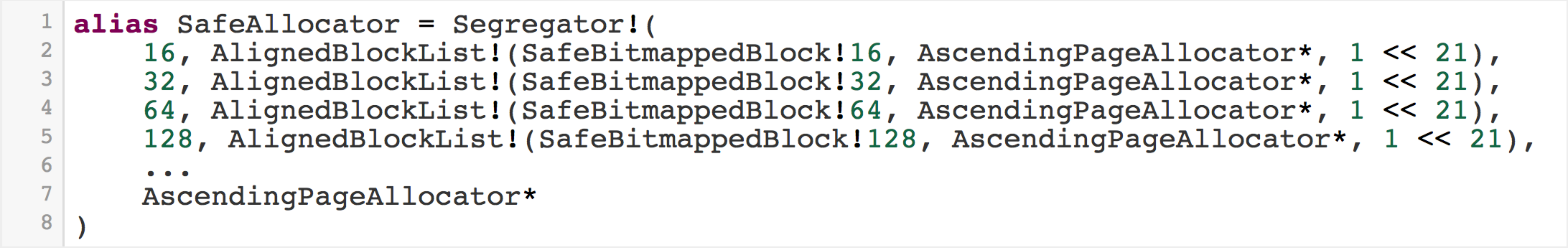
SafeAllocator v1.0



```
alias SafeAllocator = Segregator!(
    16, AlignedBlockList!(SafeBitmappedBlock!16, AscendingPageAllocator*, 1 << 21),
    32, AlignedBlockList!(SafeBitmappedBlock!32, AscendingPageAllocator*, 1 << 21),
    64, AlignedBlockList!(SafeBitmappedBlock!64, AscendingPageAllocator*, 1 << 21),
    128, AlignedBlockList!(SafeBitmappedBlock!128, AscendingPageAllocator*, 1 << 21),
    ...
    AscendingPageAllocator*
)
```

- Safe (does not reuse memory)
- High fragmentation
- Lots of page faults

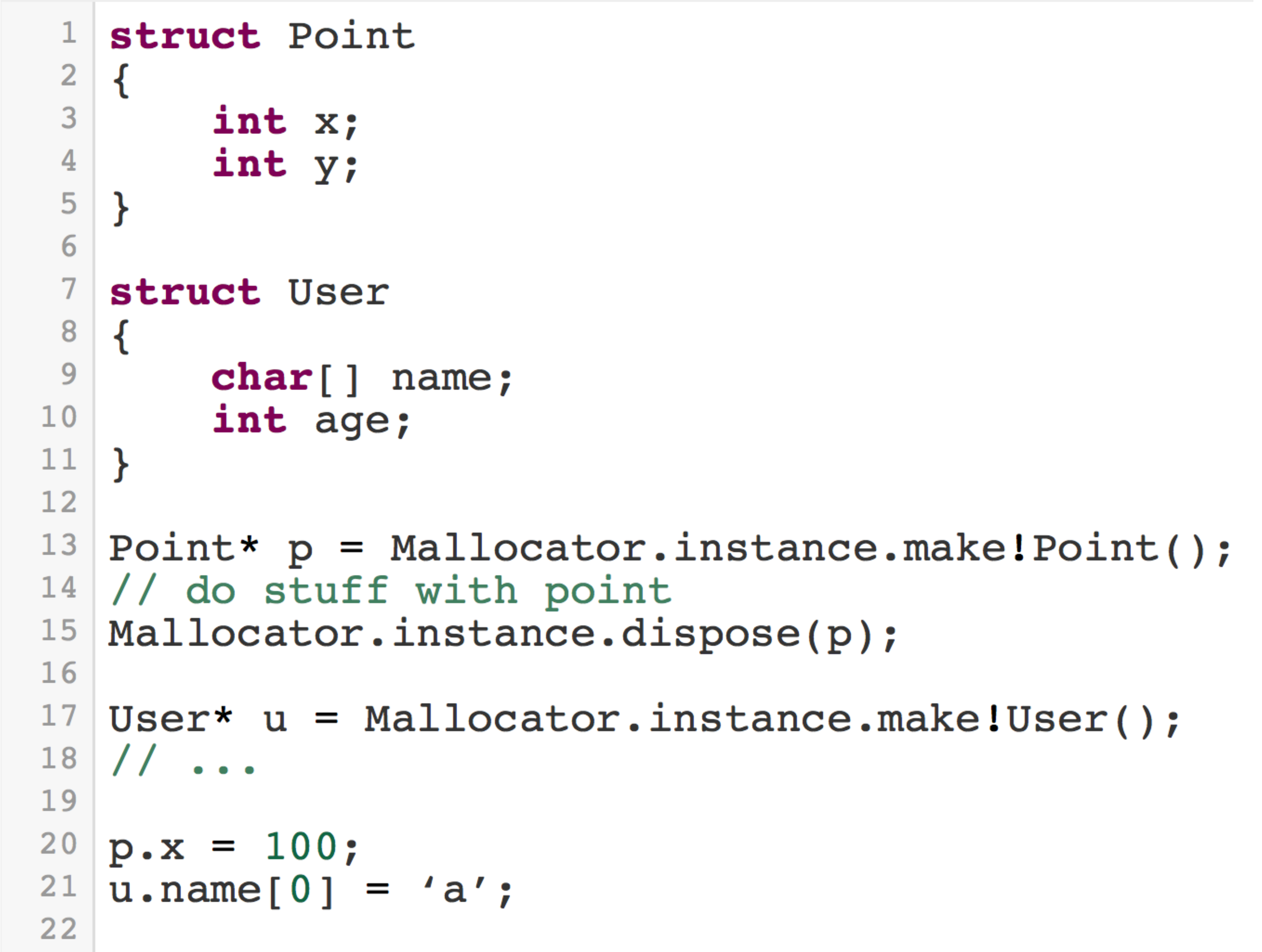
SafeAllocator v1.0



```
alias SafeAllocator = Segregator!(
    16, AlignedBlockList!(SafeBitmappedBlock!16, AscendingPageAllocator*, 1 << 21),
    32, AlignedBlockList!(SafeBitmappedBlock!32, AscendingPageAllocator*, 1 << 21),
    64, AlignedBlockList!(SafeBitmappedBlock!64, AscendingPageAllocator*, 1 << 21),
    128, AlignedBlockList!(SafeBitmappedBlock!128, AscendingPageAllocator*, 1 << 21),
    ...
    AscendingPageAllocator*
)
```

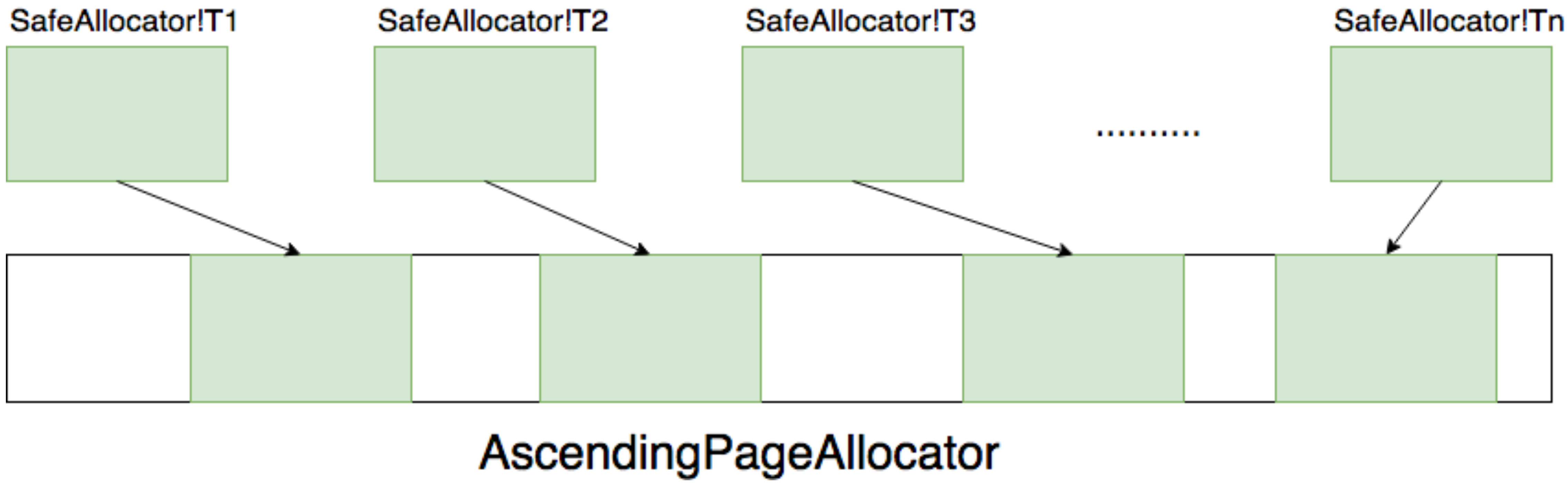
- Safe (does not reuse memory)
- High fragmentation
- Lots of page faults

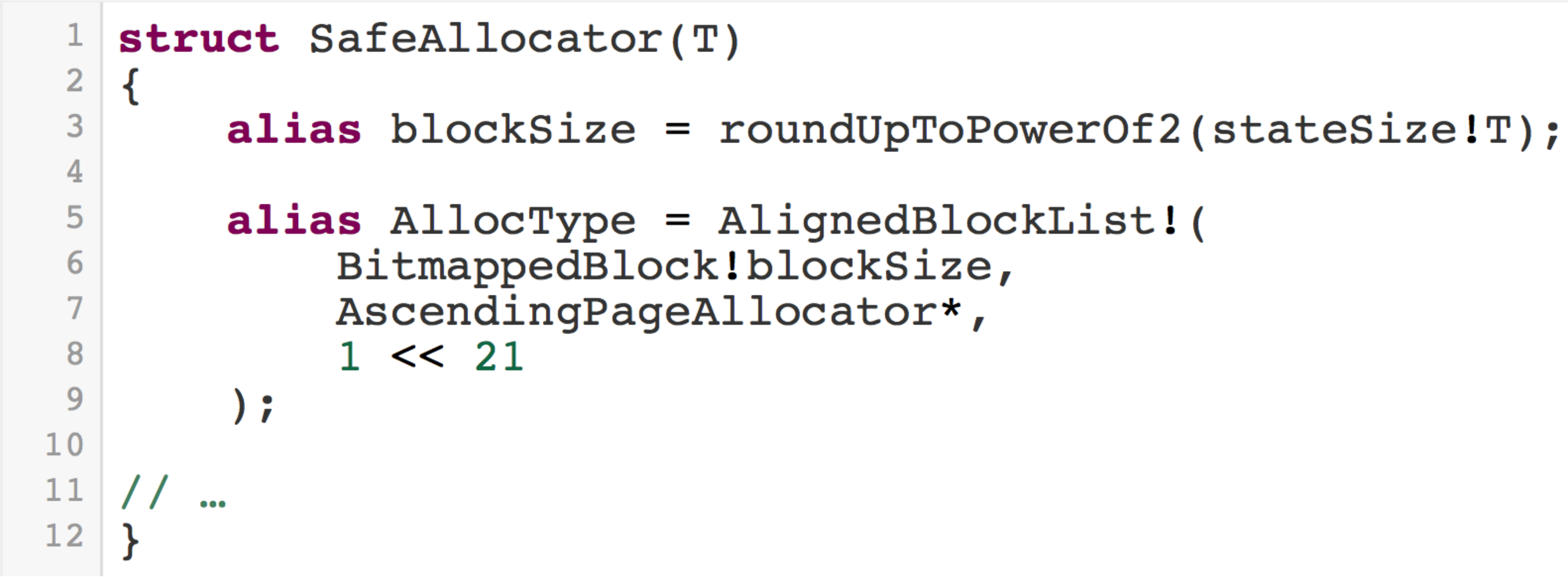
Not reusing addresses is safe but inefficient



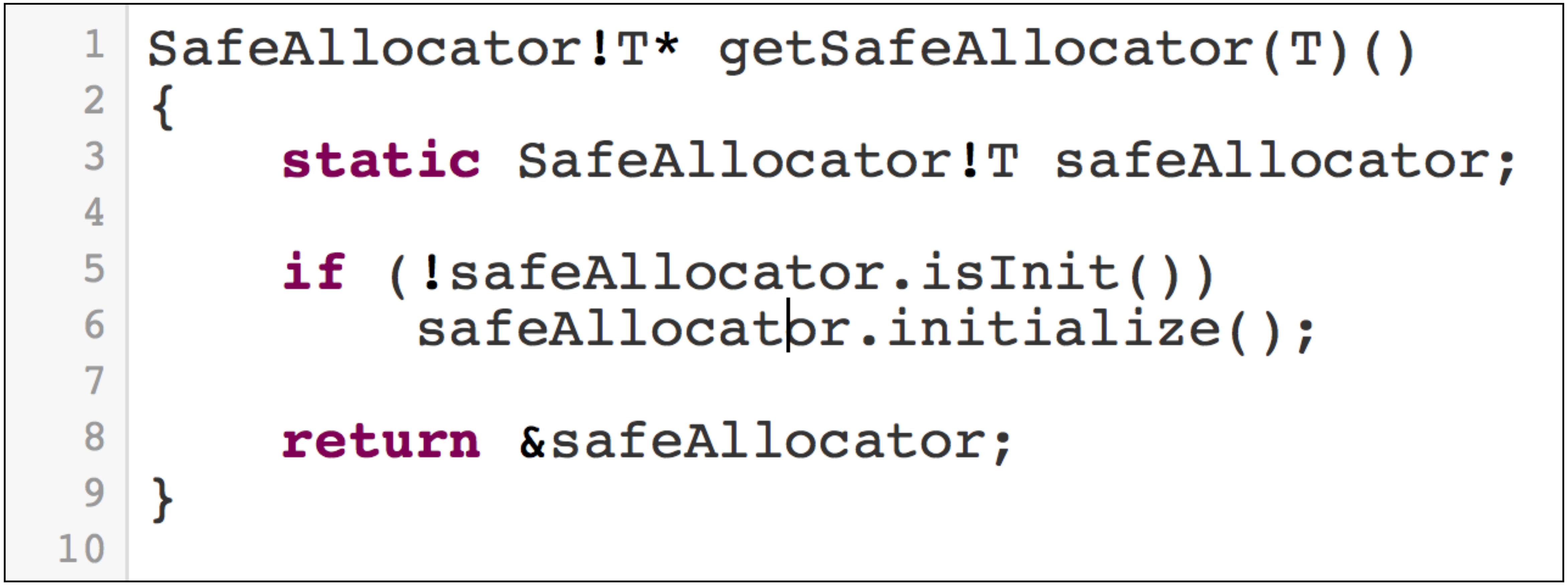
```
struct Point
{
     int x;
     int y;
}

struct User
{
     char[] name;
     int age;
}

Point* p = Mallocator.instance.make!Point();
// do stuff with point
Mallocator.instance.dispose(p);

User* u = Mallocator.instance.make!User();
// ...

p.x = 100;
u.name[0] = ‘a’;

```

Reuse memory only for the same types

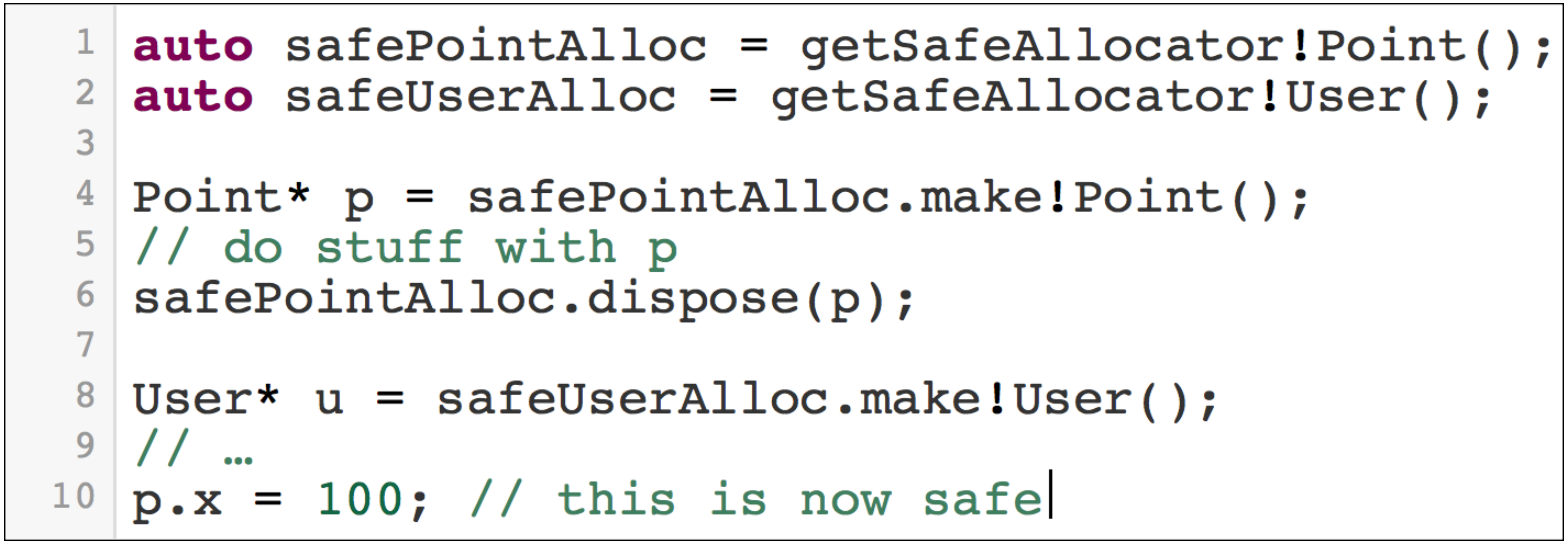




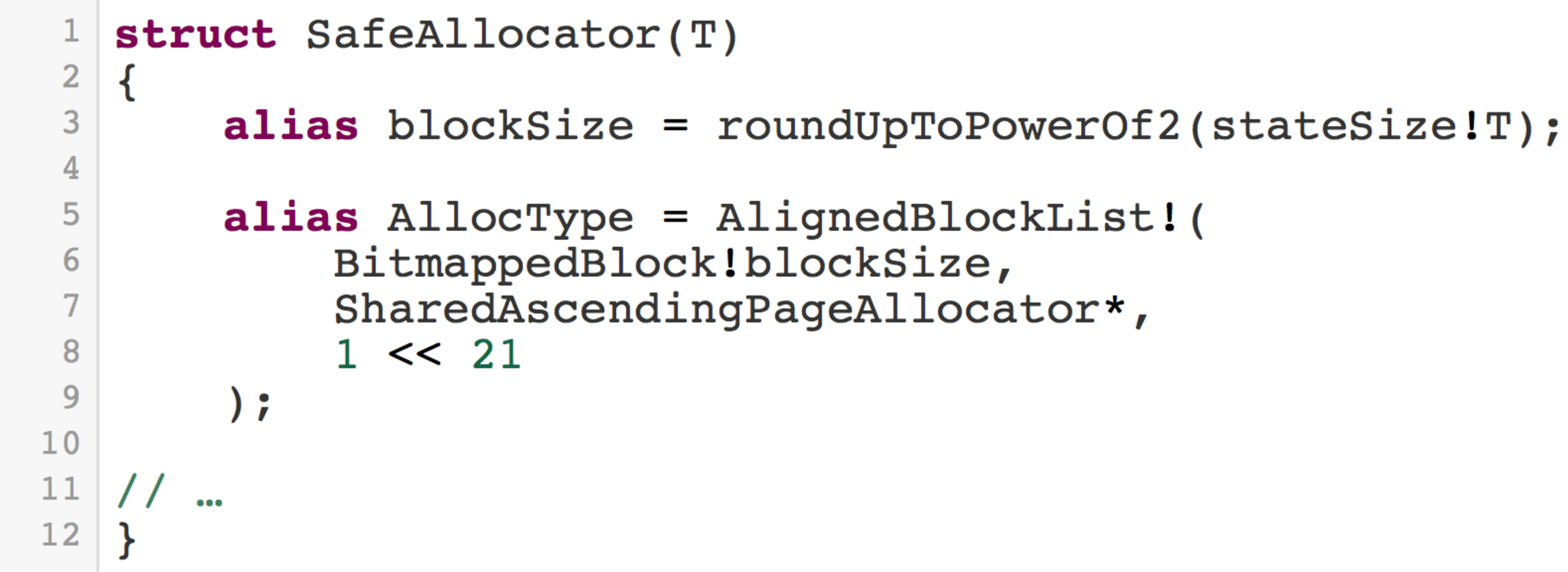
```
struct SafeAllocator(T)
{
    alias blockSize = roundUpToPowerOf2(stateSize!T);

    alias AllocType = AlignedBlockList!(
        BitmappedBlock!blockSize,
        AscendingPageAllocator*,
        1 << 21
    );

// …
}
```



```
SafeAllocator!T* getSafeAllocator(T)()
{
    static SafeAllocator!T safeAllocator;

    if (!safeAllocator.isInit())
        safeAllocator.initialize();

    return &safeAllocator;
}

```



```
auto safePointAlloc = getSafeAllocator!Point();
auto safeUserAlloc = getSafeAllocator!User();

Point* p = safePointAlloc.make!Point();
// do stuff with p
safePointAlloc.dispose(p);

User* u = safeUserAlloc.make!User();
// …
p.x = 100; // this is now safe
```



```
struct SafeAllocator(T)
{
    alias blockSize = roundUpToPowerOf2(stateSize!T);

    alias AllocType = AlignedBlockList!(
        BitmappedBlock!blockSize,
        SharedAscendingPageAllocator*,
        1 << 21
    );

// …
}
```

SafeAllocator v2.0

Safe

High performance

Scalable

SafeAllocator v2.0

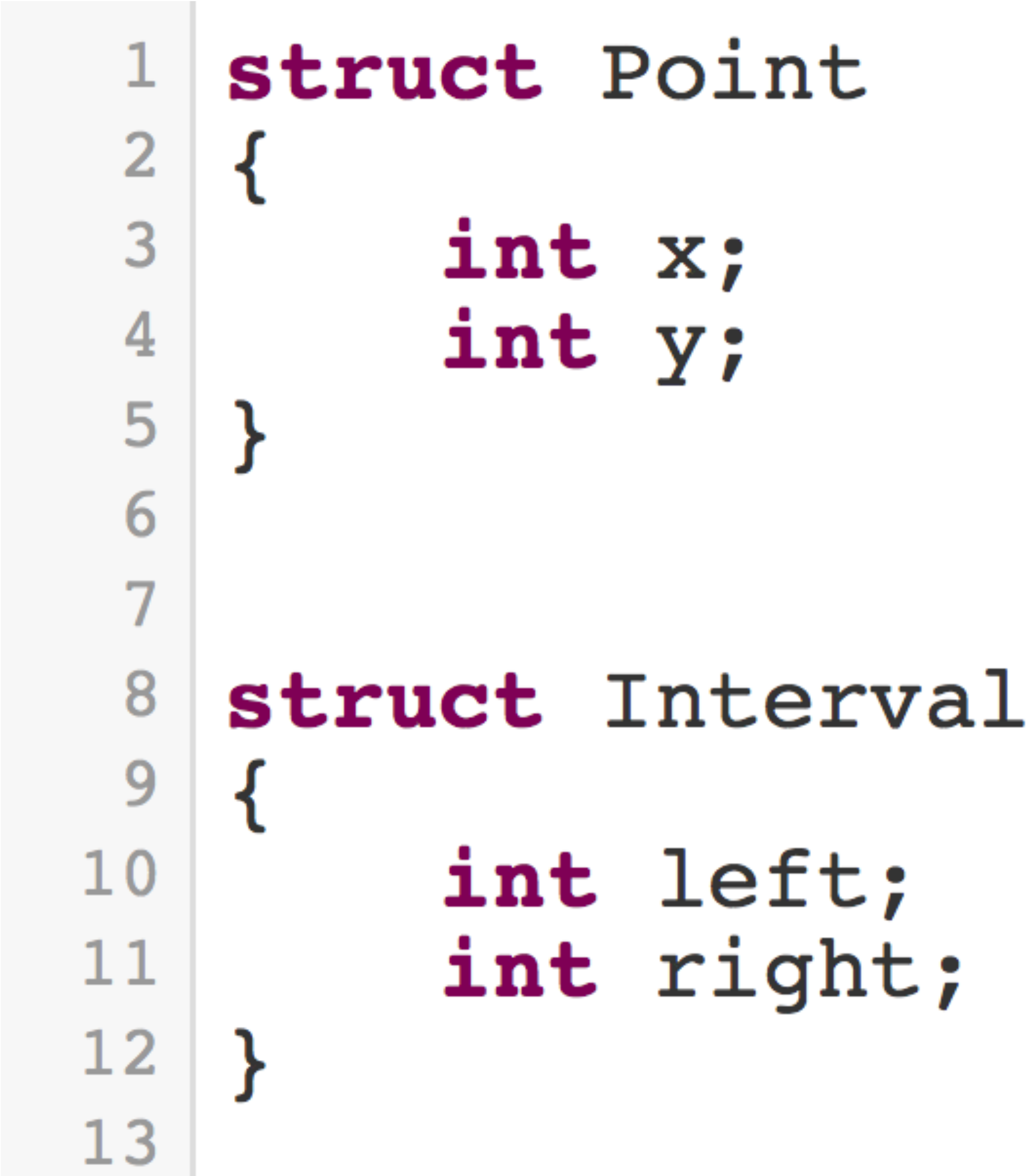
Safe

High performance

Scalable

Many types = high fragmentation

THIS ISN'T EVEN MY FINAL FORM.
made on imgur



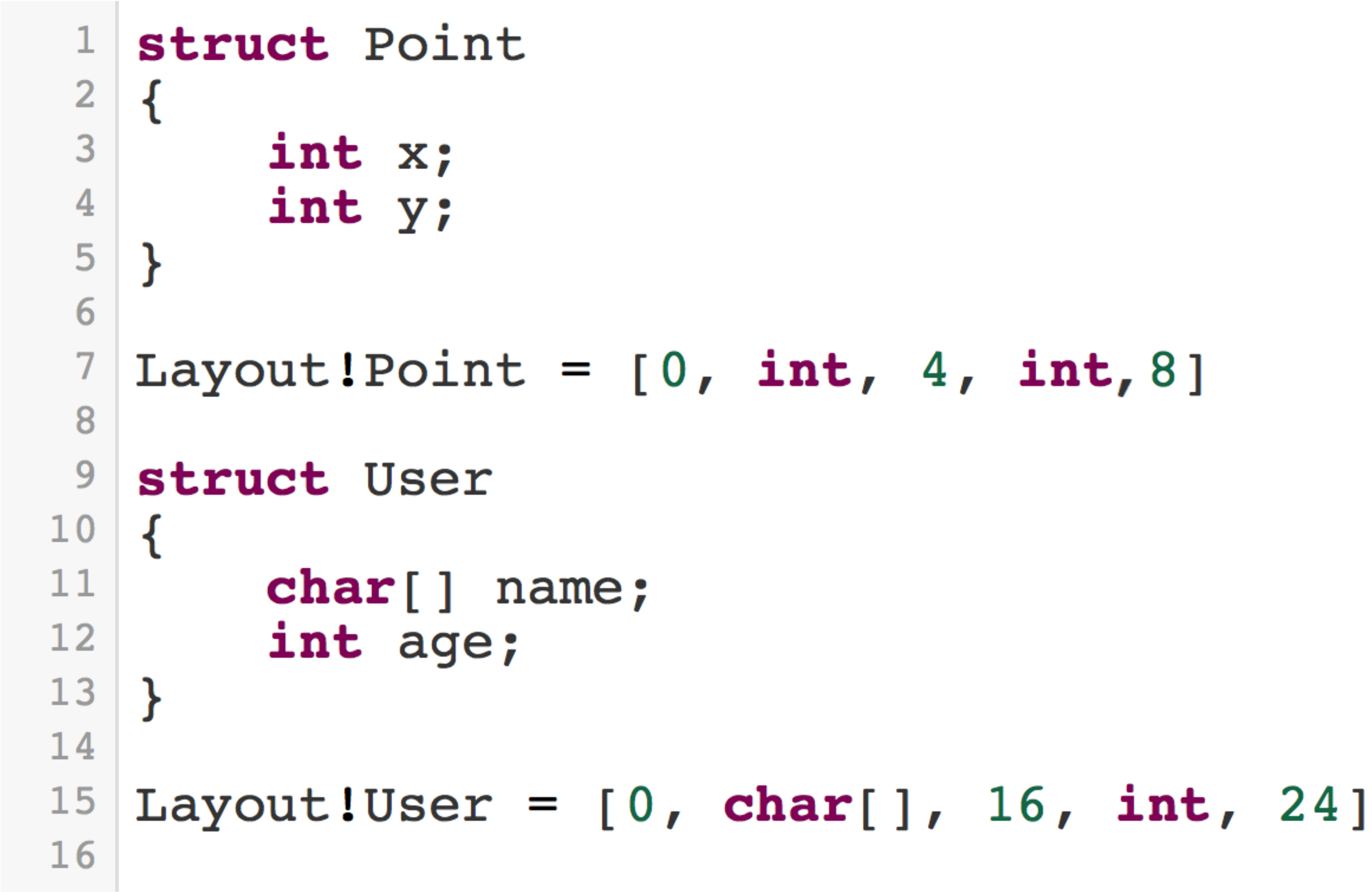
```
struct Point
{
    int x;
    int y;
}


struct Interval
{
    int left;
    int right;
}

```

Memory layout

Many types = High fragmentation

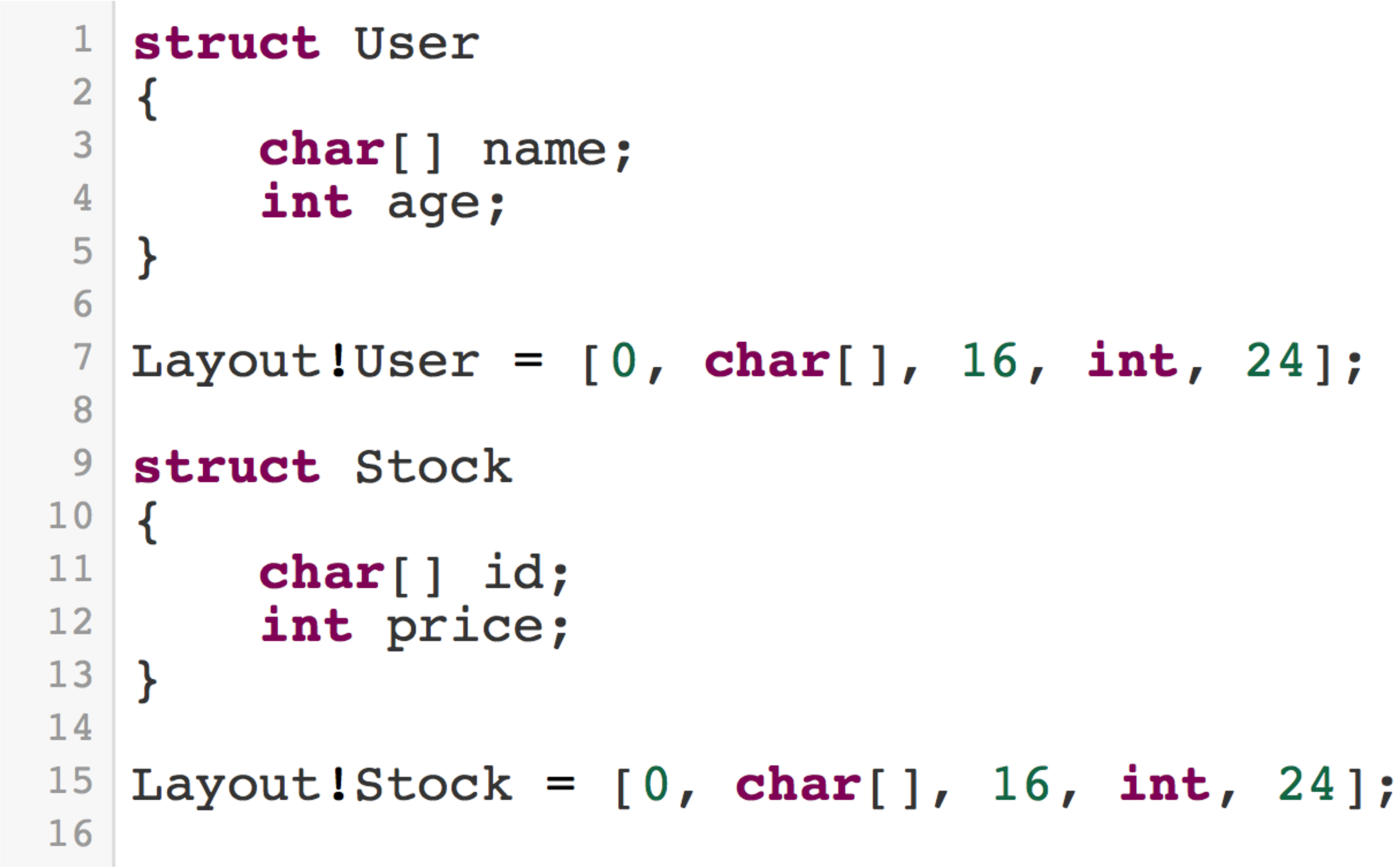
Reuse memory among types with the same memory layout

Layout



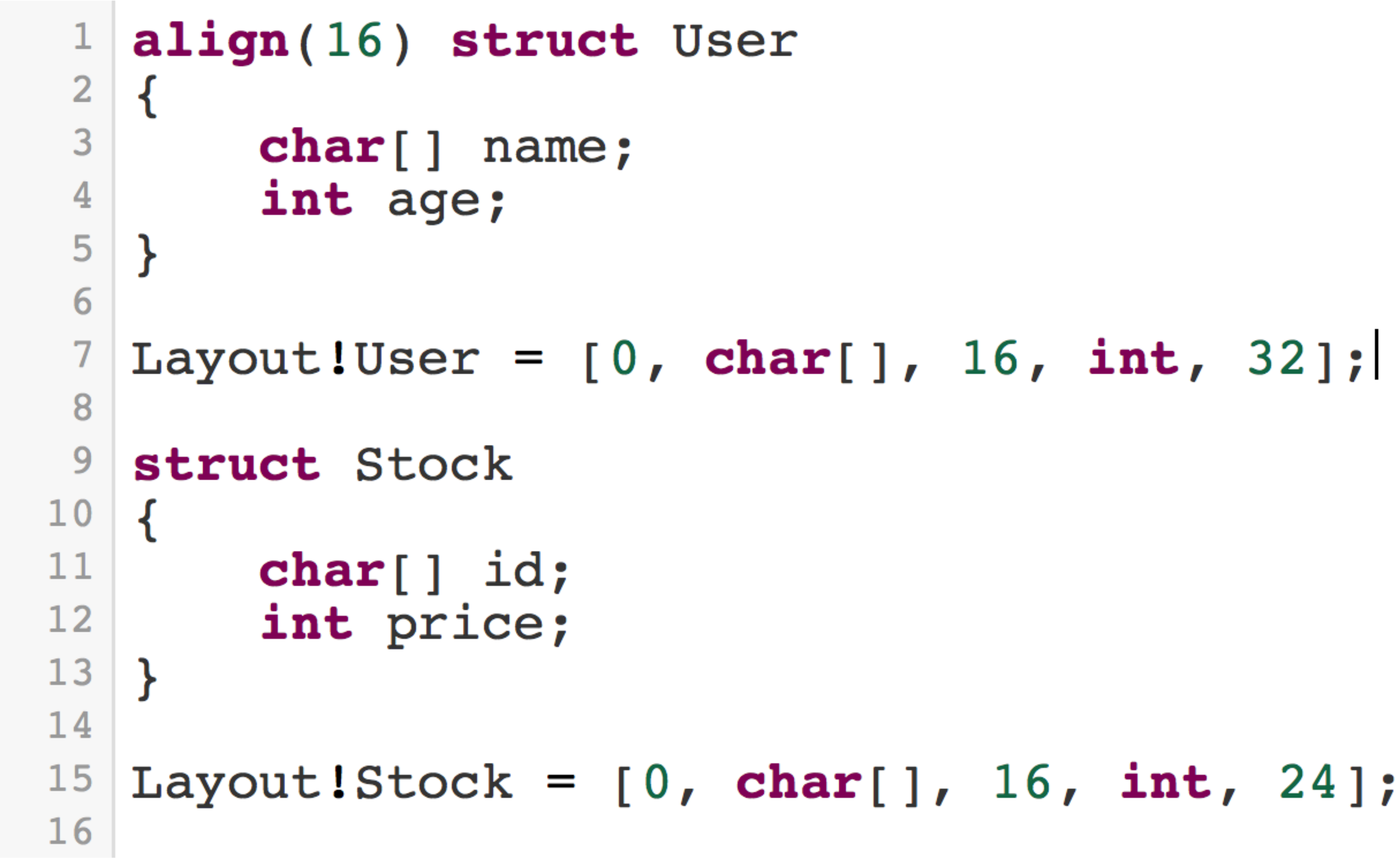
```
struct Point
{
	int x;
	int y;
}

Layout!Point = [0, int, 4, int,8]

struct User
{
	char[] name;
	int age;
}

Layout!User = [0, char[], 16, int, 24]

```

Layout



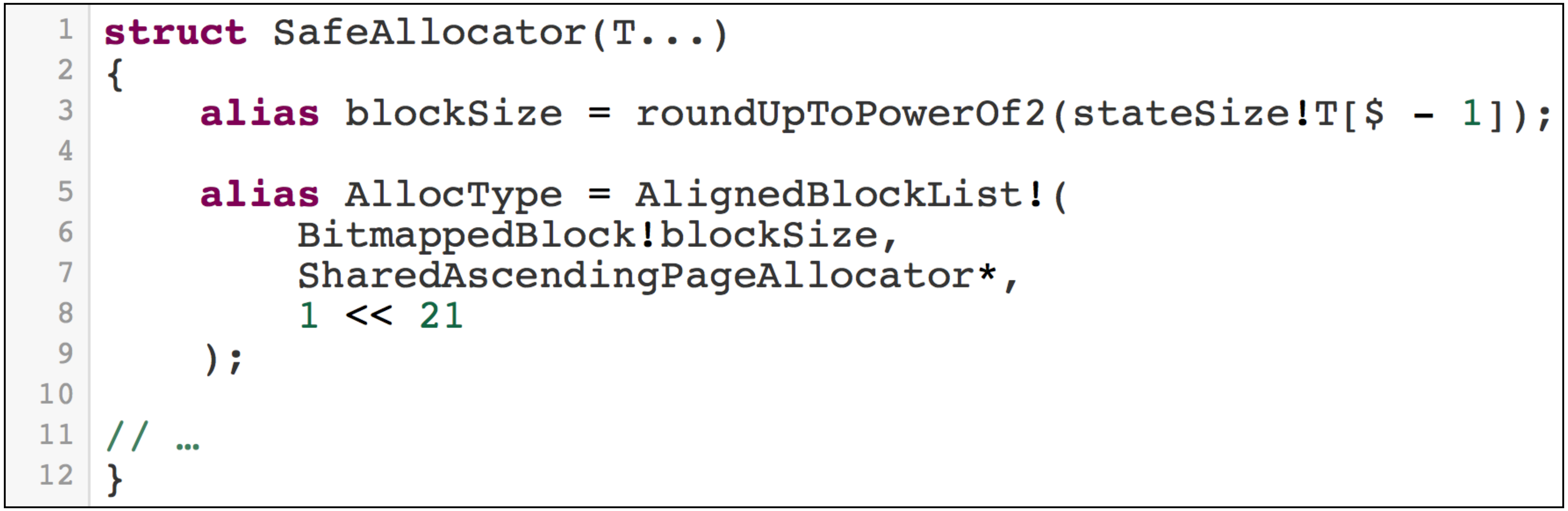
```
struct User
{
	char[] name;
	int age;
}

Layout!User = [0, char[], 16, int, 24];

struct Stock
{
	char[] id;
	int price;
}

Layout!Stock = [0, char[], 16, int, 24];
```

Layout

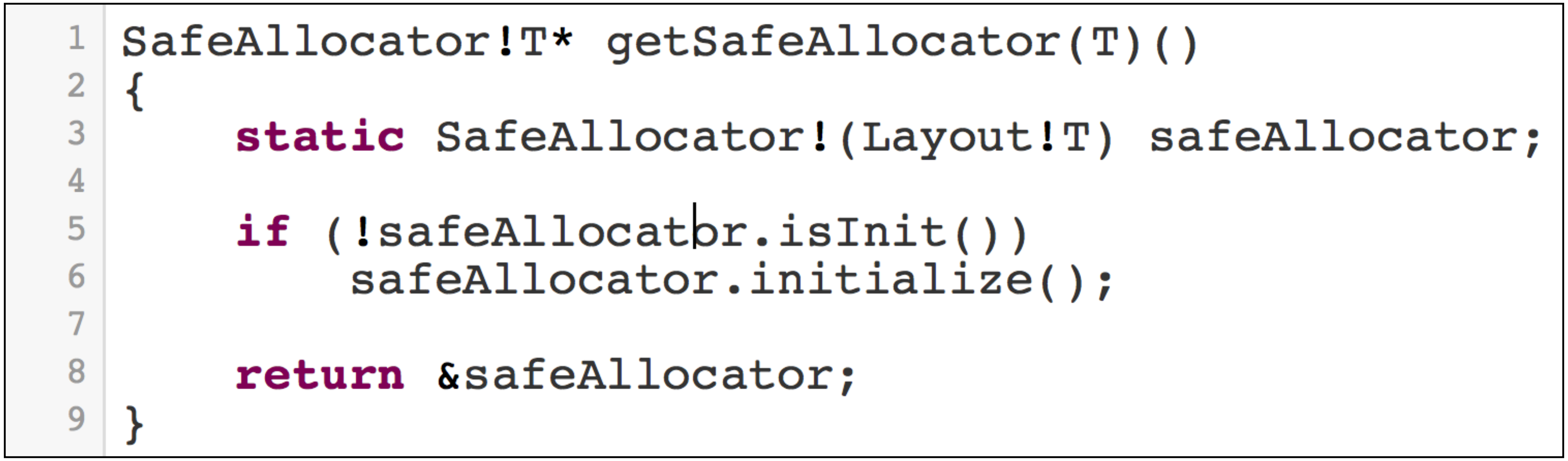


```
align(16) struct User
{
	char[] name;
	int age;
}

Layout!User = [0, char[], 16, int, 32];

struct Stock
{
	char[] id;
	int price;
}

Layout!Stock = [0, char[], 16, int, 24];
```

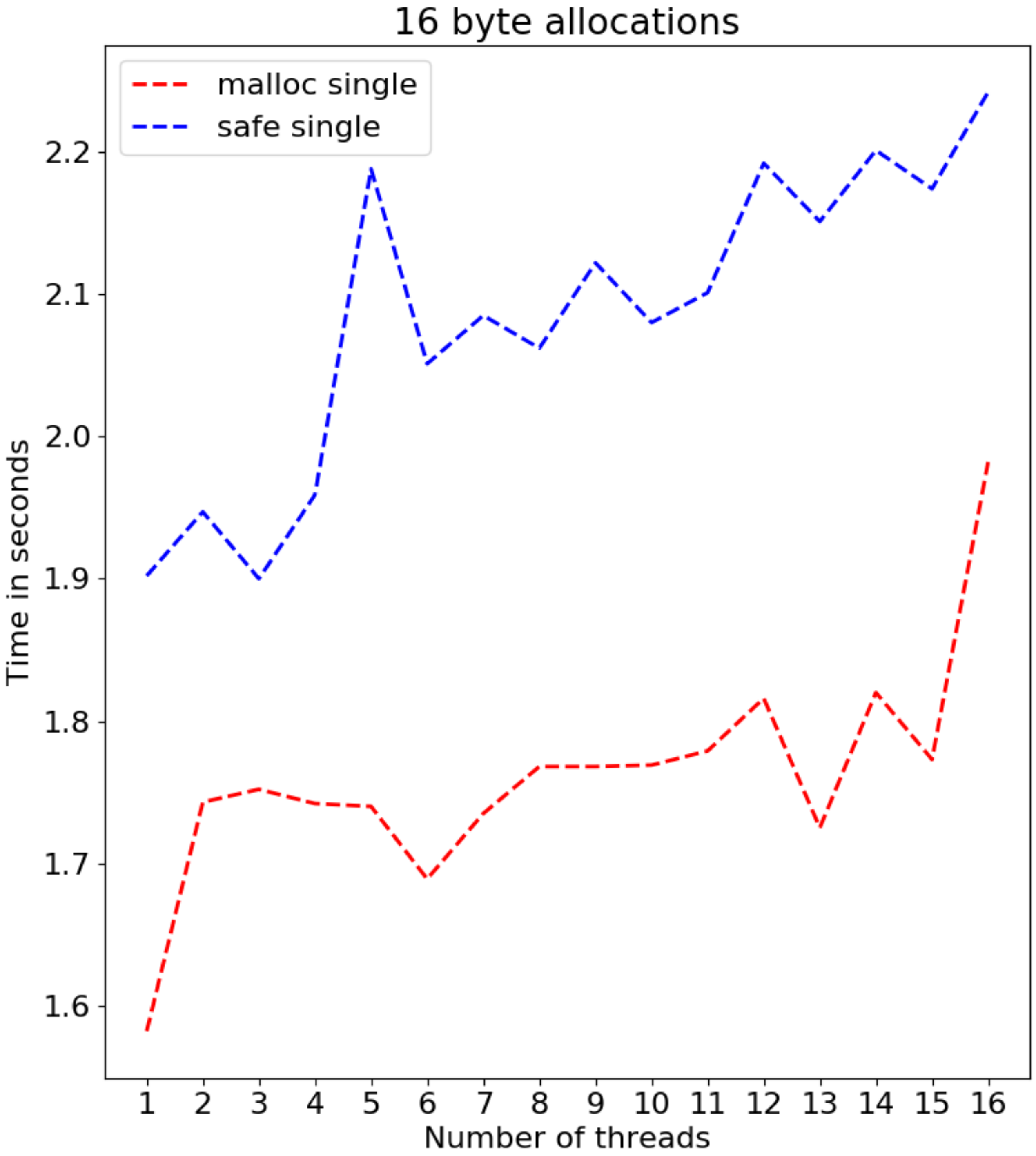
SafeAllocator v3.0

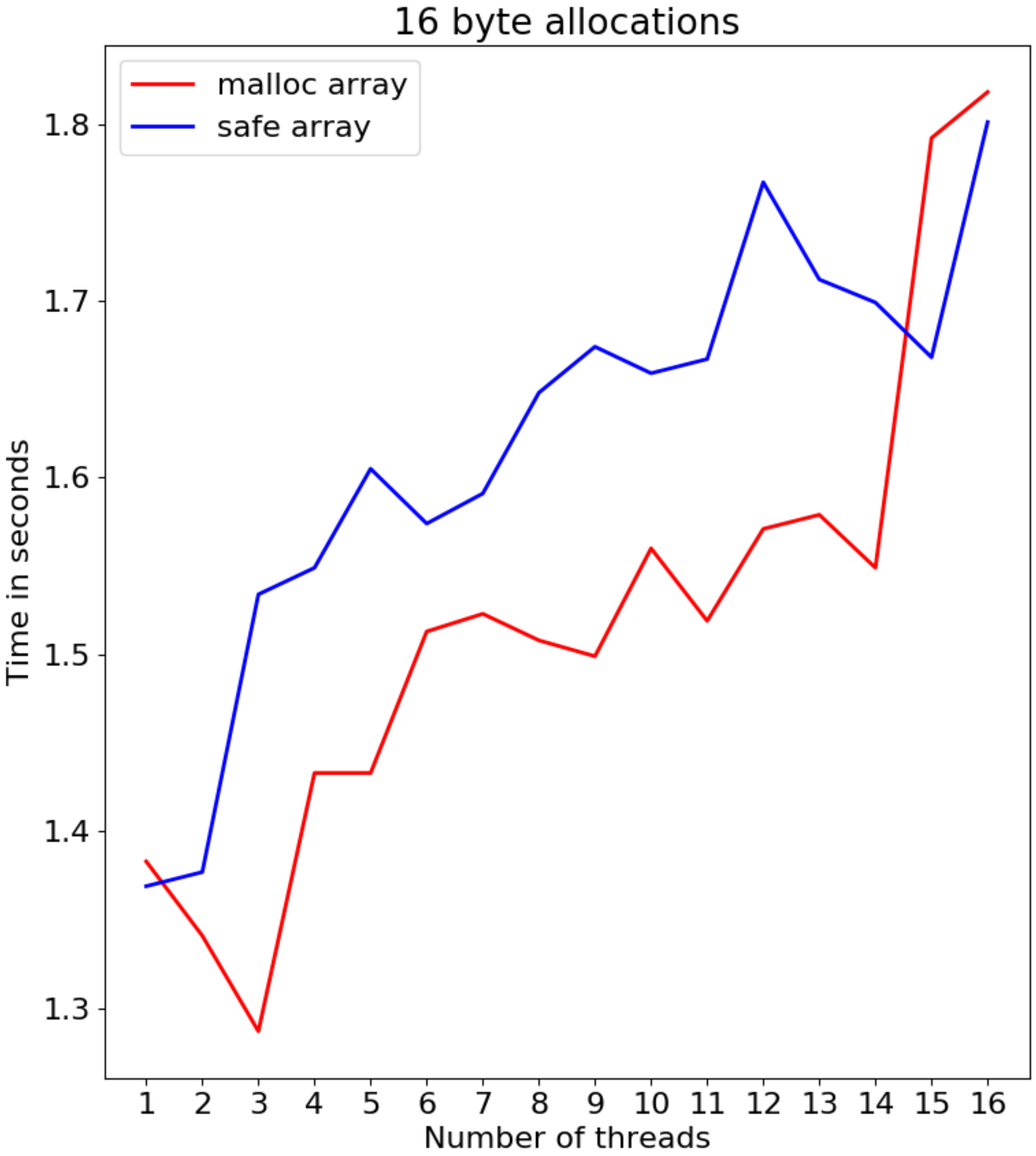


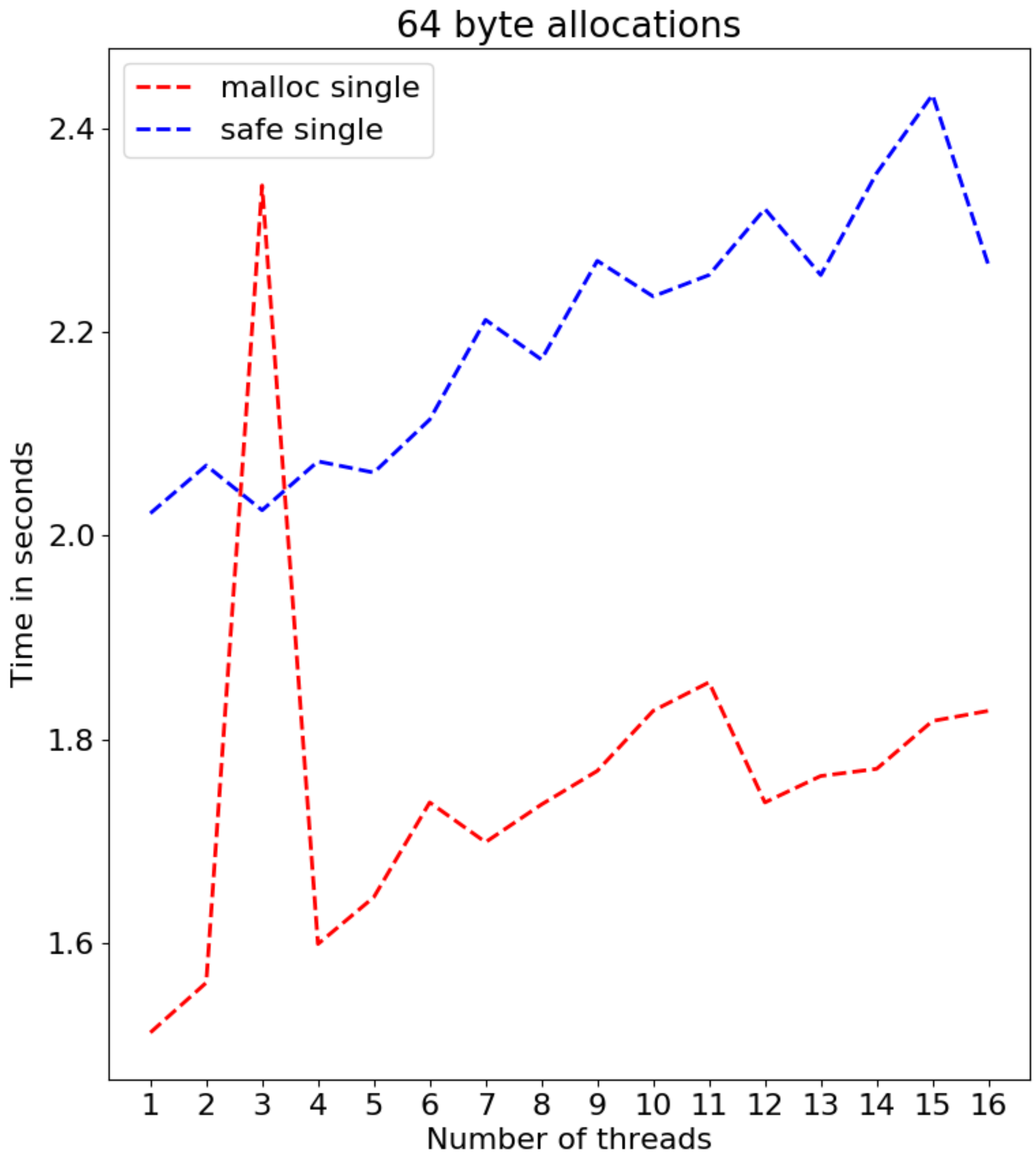
```
struct SafeAllocator(T...)
{
    alias blockSize = roundUpToPowerOf2(stateSize!T[$ - 1]);

    alias AllocType = AlignedBlockList!(
        BitmappedBlock!blockSize,
        SharedAscendingPageAllocator*,
        1 << 21
    );

// …
}
```

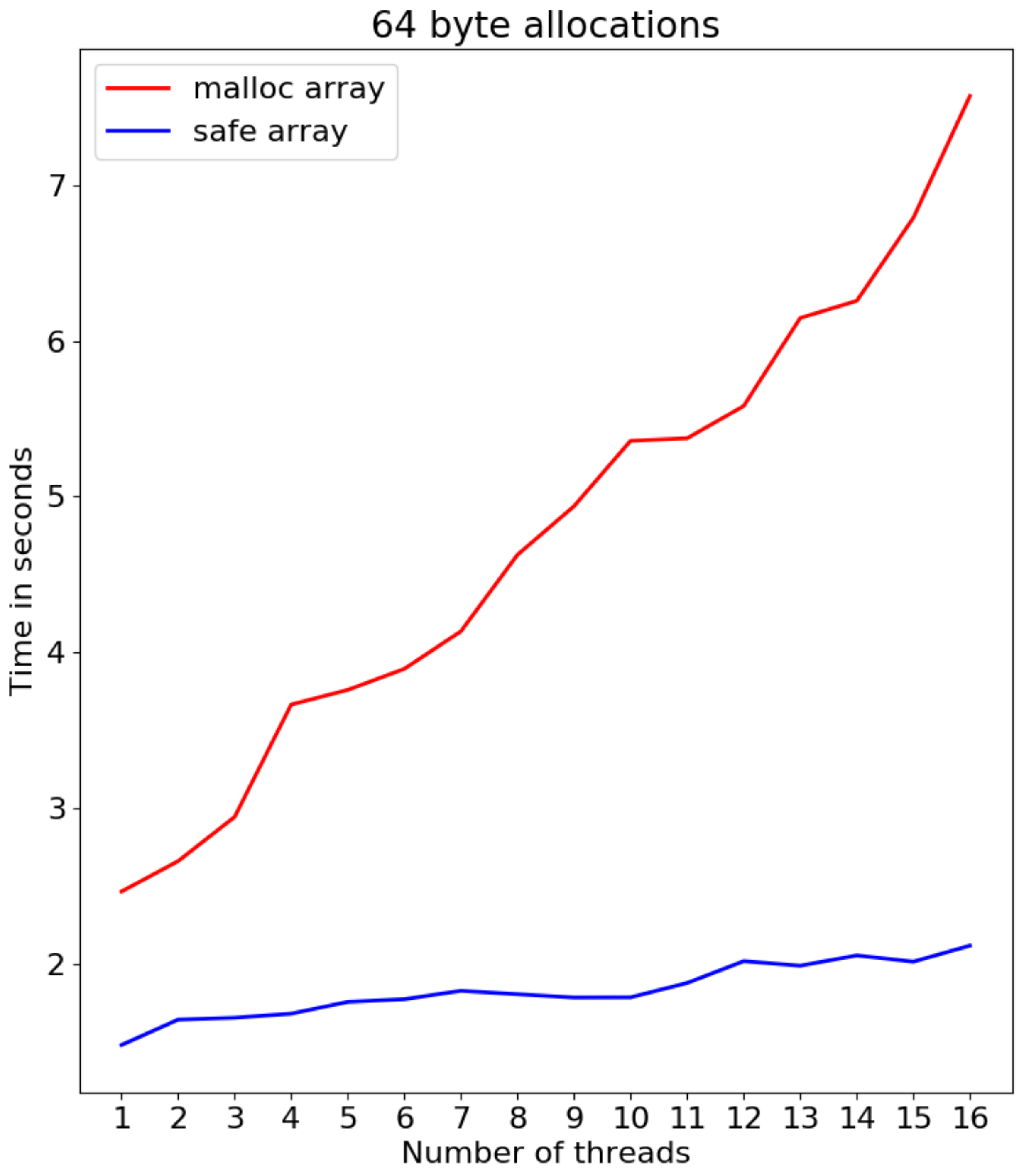


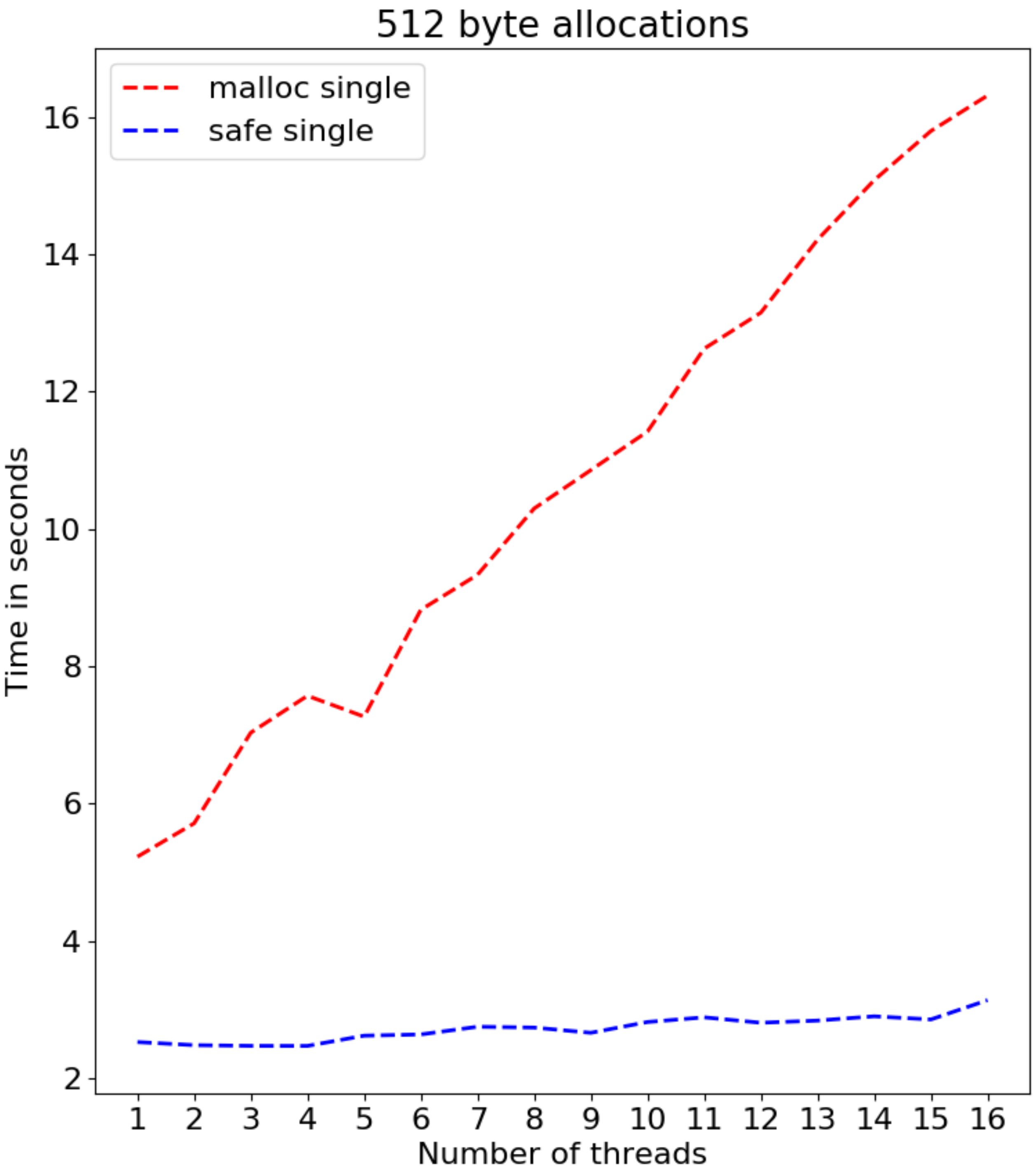
```
SafeAllocator!T* getSafeAllocator(T)()
{
    static SafeAllocator!(Layout!T) safeAllocator;

    if (!safeAllocator.isInit())
        safeAllocator.initialize();

    return &safeAllocator;
}
```

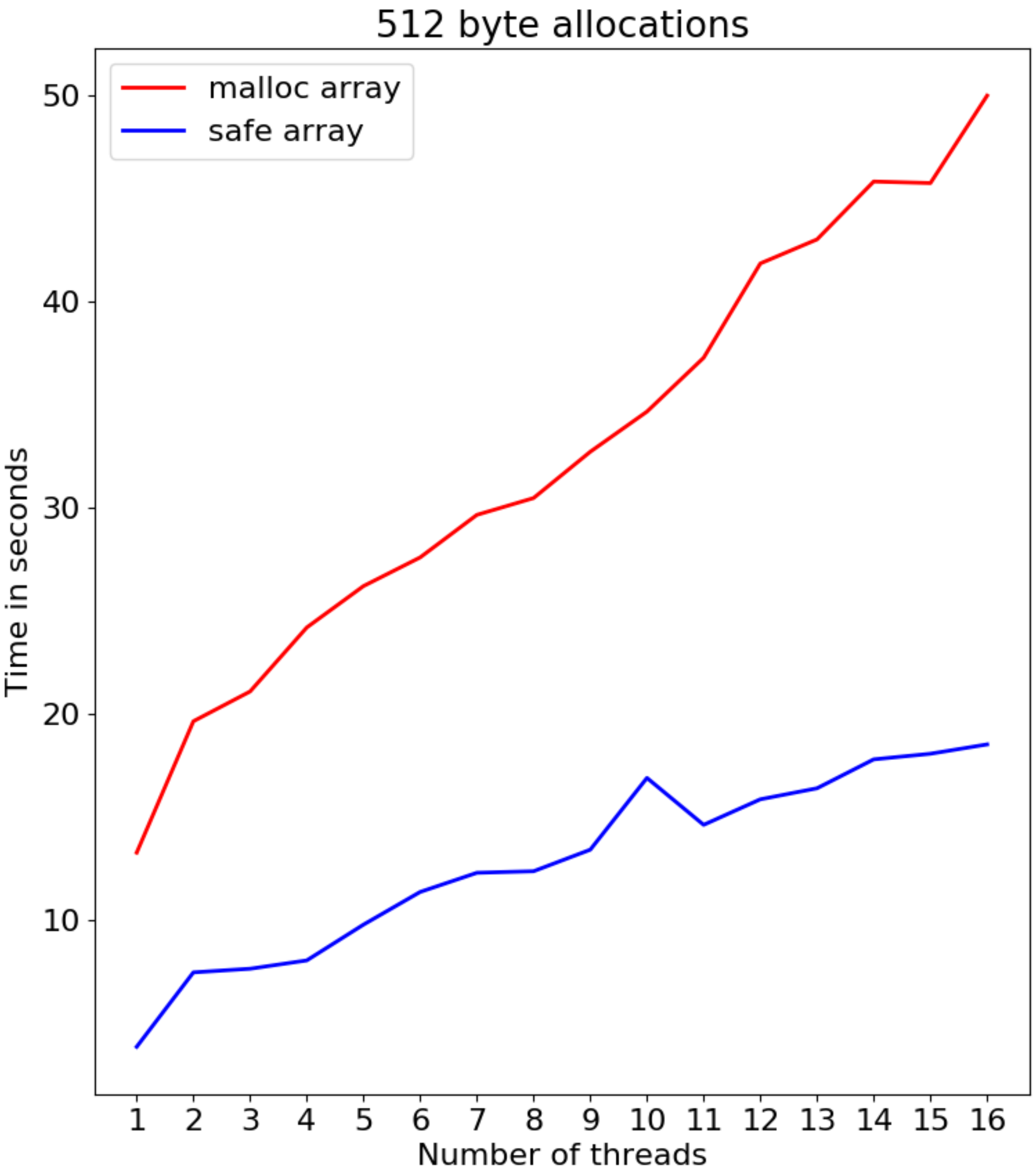

16 byte allocations
malloc single
safe single
Time in seconds
1.6
1.7
1.8
1.9
2.0
2.1
2.2
1 2 3 4 5 6 7 8 9 10 11 12 13 14 15 16
Number of threads


16 byte allocations
malloc array
safe array
Time in seconds
1.3
1.4
1.5
1.6
1.7
1.8
1 2 3 4 5 6 7 8 9 10 11 12 13 14 15 16
Number of threads


64 byte allocations
malloc single
safe single
Time in seconds
2.4
2.2
2.0
1.8
1.6
1
2
3
4
5
6
7
8
9
10
11
12
13
14
15
16
Number of threads


64 byte allocations
malloc array
safe array
Time in seconds
7
6
5
4
3
2
1
2
3
4
5
6
7
8
9
10
11
12
13
14
15
16
Number of threads


512 byte allocations
malloc single
safe single
Time in seconds
2
4
6
8
10
12
14
16
1
2
3
4
5
6
7
8
9
10
11
12
13
14
15
16
Number of threads


512 byte allocations
malloc array
safe array
Time in seconds
10
20
30
40
50
1
2
3
4
5
6
7
8
9
10
11
12
13
14
15
16
Number of threads

Summary

	SAFETY	PERFORMANCE	FRAGMENTATION
SAFE ALLOCATOR V1.0	Not reuse memory	Fast for small and large allocations	High
SAFE ALLOCATOR V2.0	Type reuse	Fast	Moderate
SAFE ALLOCATOR V3.0	Layout reuse	Fast	Low

Future work

Improve documentation for current allocators

Improve specification for Layout and SafeAllocator

More benchmarks

Acknowledgments