

stack.h

```
typedef struct {  
    void *elems;  
    int elemSize;  
    int logLength;  
    int allocLength;  
} stack;
```

```
stack.h
```

```
typedef struct {  
    void *elems;  
    int elemSize;  
    int logLength;  
    int allocLength;  
} stack;
```

Writing a generic container in pure C is hard, and it's hard for two reasons:

- 1. The language doesn't offer any real support for encapsulation or information hiding. That means that the data structures expose information about internal representation right there in the interface file for everyone to see and manipulate. The best we can do is document that the data structure should be treated as an abstract data type, and that the client shouldn't directly manage the fields. Instead, he should just rely on the functions provided to manage the internals for him.*
- 2. C doesn't allow data types to be passed as parameters. That means a generic container needs to manually manage memory in terms of the client element size, not client data type. This translates to a bunch of malloc, realloc, free, memcpy, and memmove calls involving void *s. This is the very type of programming that makes C difficult.*

stack.h

```
typedef struct {
    void *elems;
    int elemSize;
    int logLength;
    int allocLength;
} stack;

void StackNew(stack *s, int elemSize);
void StackDispose(stack *s);
bool StackEmpty(const stack *s);
void StackPush(stack *s, const void *elemAddr);
void StackPop(stack *s, void *elemAddr);
```

This is what you see as a client.

```
stack.h
```

```
typedef struct {  
    void *elems;  
    int elemSize;  
    int logLength;  
    int allocLength;  
} stack;
```

```
void StackNew(stack *s, int elemSize);  
void StackDispose(stack *s);  
bool StackEmpty(const stack *s);  
void StackPush(stack *s, const void *elemAddr);  
void StackPop(stack *s, void *elemAddr);
```

This is what you see as a client, but as a client, you should ignore the internals of an exposed data structure unless the documentation explicitly says otherwise.

stack.h

```
typedef struct {
    void *elems;
    int elemSize;
    int logLength;
    int allocLength;
} stack;

void StackNew(stack *s, int elemSize);
void StackDispose(stack *s);
bool StackEmpty(const stack *s);
void StackPush(stack *s, const void *elemAddr);
void StackPop(stack *s, void *elemAddr);
```

client.c

```
int main(int argc, char *argv[])
{
    int val;
    stack intStack;

    StackNew(&intStack, sizeof(int));
    for (val = 0; val < 6; val++)
        StackPush(&intStack, &val);

    while (!StackEmpty(&intStack)) {
        StackPop(&intStack, &val);
        printf("This just popped: %d\n", val);
    }
    StackDispose(&intStack);
}
```

client.c

```
int main(int argc, char *argv[])
{
    int val;
    stack intStack;

    StackNew(&intStack, sizeof(int));
    for (val = 0; val < 6; val++)
        StackPush(&intStack, &val);

    while (!StackEmpty(&intStack)) {
        StackPop(&intStack, &val);
        printf("This just popped: %d\n", val);
    }
    StackDispose(&intStack);
}
```

client.c

```
int main(int argc, char *argv[])
{
    int val;
    stack intStack;

    StackNew(&intStack, sizeof(int));
    for (val = 0; val < 6; val++)
        StackPush(&intStack, &val);

    while (!StackEmpty(&intStack)) {
        StackPop(&intStack, &val);
        printf("This just popped: %d\n", val);
    }
    StackDispose(&intStack);
}
```

val

client.c

```
int main(int argc, char *argv[])
{
    int val;
    stack intStack;

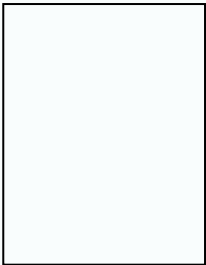
    StackNew(&intStack, sizeof(int));
    for (val = 0; val < 6; val++)
        StackPush(&intStack, &val);

    while (!StackEmpty(&intStack)) {
        StackPop(&intStack, &val);
        printf("This just popped: %d\n", val);
    }
    StackDispose(&intStack);
}
```

val



intStack

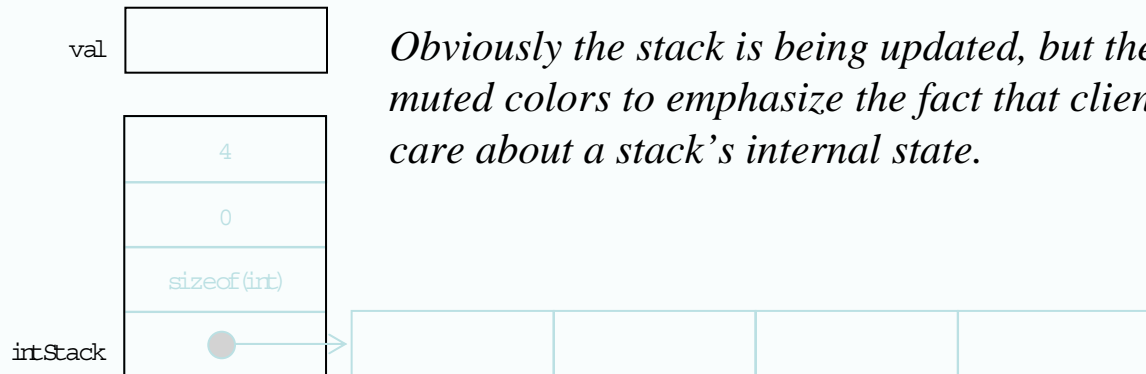


client.c

```
int main(int argc, char *argv[])
{
    int val;
    stack intStack;

    StackNew(&intStack, sizeof(int));
    for (val = 0; val < 6; val++)
        StackPush(&intStack, &val);

    while (!StackEmpty(&intStack)) {
        StackPop(&intStack, &val);
        printf("This just popped: %d\n", val);
    }
    StackDispose(&intStack);
}
```



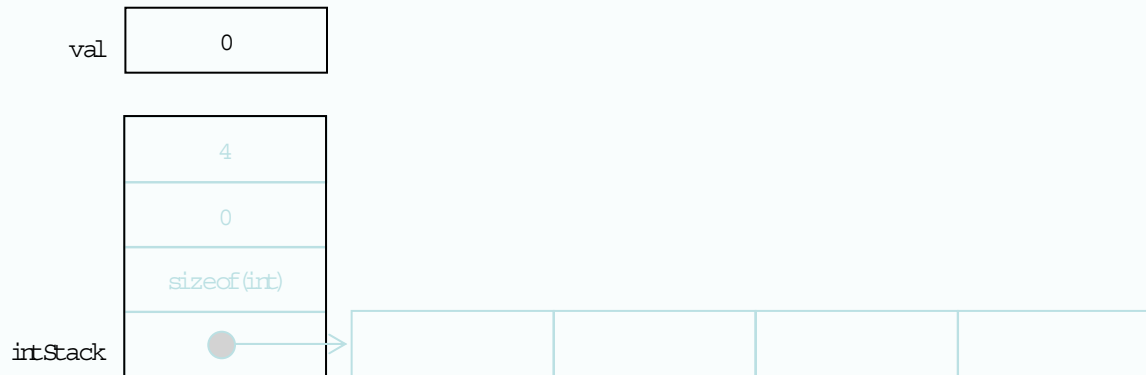
Obviously the stack is being updated, but the drawings come in pale, muted colors to emphasize the fact that clients shouldn't really know or care about a stack's internal state.

client.c

```
int main(int argc, char *argv[])
{
    int val;
    stack intStack;

    StackNew(&intStack, sizeof(int));
    for (val = 0; val < 6; val++)
        StackPush(&intStack, &val);

    while (!StackEmpty(&intStack)) {
        StackPop(&intStack, &val);
        printf("This just popped: %d\n", val);
    }
    StackDispose(&intStack);
}
```



client.c

```
int main(int argc, char *argv[])
{
    int val;
    stack intStack;

    StackNew(&intStack, sizeof(int));
    for (val = 0; val < 6; val++)
        StackPush(&intStack, &val);

    while (!StackEmpty(&intStack)) {
        StackPop(&intStack, &val);
        printf("This just popped: %d\n", val);
    }
    StackDispose(&intStack);
}
```



client.c

```
int main(int argc, char *argv[])
{
    int val;
    stack intStack;

    StackNew(&intStack, sizeof(int));
    for (val = 0; val < 6; val++)
        StackPush(&intStack, &val);

    while (!StackEmpty(&intStack)) {
        StackPop(&intStack, &val);
        printf("This just popped: %d\n", val);
    }
    StackDispose(&intStack);
}
```



client.c

```
int main(int argc, char *argv[])
{
    int val;
    stack intStack;

    StackNew(&intStack, sizeof(int));
    for (val = 0; val < 6; val++)
        StackPush(&intStack, &val);

    while (!StackEmpty(&intStack)) {
        StackPop(&intStack, &val);
        printf("This just popped: %d\n", val);
    }
    StackDispose(&intStack);
}
```



client.c

```
int main(int argc, char *argv[])
{
    int val;
    stack intStack;

    StackNew(&intStack, sizeof(int));
    for (val = 0; val < 6; val++)
        StackPush(&intStack, &val);

    while (!StackEmpty(&intStack)) {
        StackPop(&intStack, &val);
        printf("This just popped: %d\n", val);
    }
    StackDispose(&intStack);
}
```

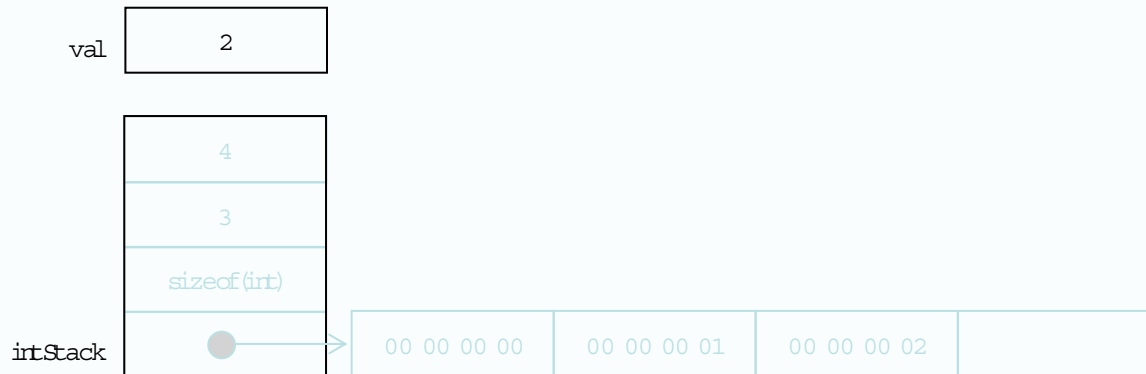


client.c

```
int main(int argc, char *argv[])
{
    int val;
    stack intStack;

    StackNew(&intStack, sizeof(int));
    for (val = 0; val < 6; val++)
        StackPush(&intStack, &val);

    while (!StackEmpty(&intStack)) {
        StackPop(&intStack, &val);
        printf("This just popped: %d\n", val);
    }
    StackDispose(&intStack);
}
```

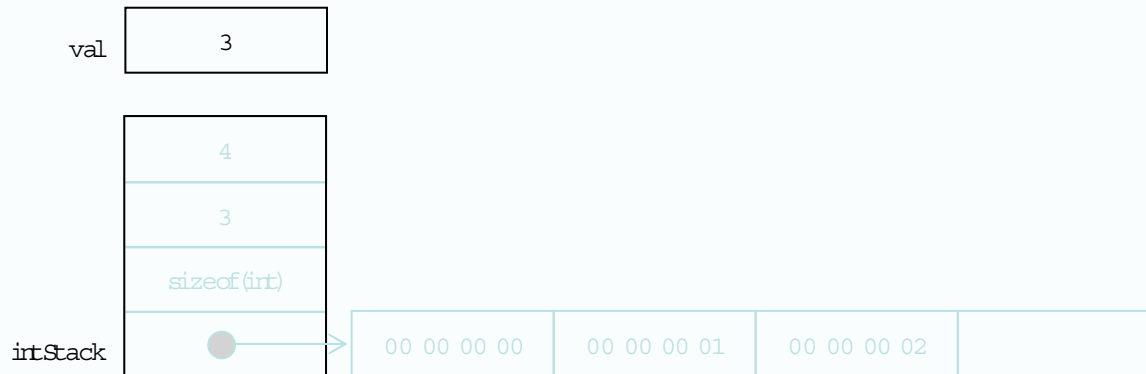


client.c

```
int main(int argc, char *argv[])
{
    int val;
    stack intStack;

    StackNew(&intStack, sizeof(int));
    for (val = 0; val < 6; val++)
        StackPush(&intStack, &val);

    while (!StackEmpty(&intStack)) {
        StackPop(&intStack, &val);
        printf("This just popped: %d\n", val);
    }
    StackDispose(&intStack);
}
```

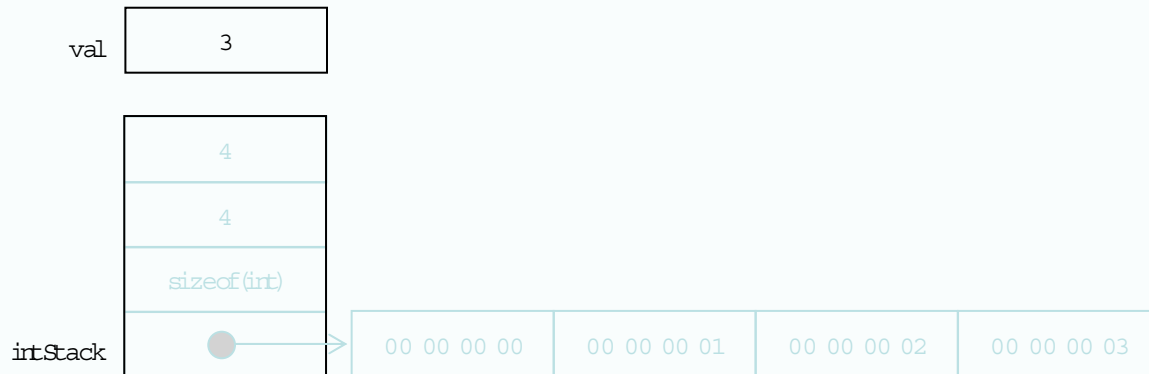


client.c

```
int main(int argc, char *argv[])
{
    int val;
    stack intStack;

    StackNew(&intStack, sizeof(int));
    for (val = 0; val < 6; val++)
        StackPush(&intStack, &val);

    while (!StackEmpty(&intStack)) {
        StackPop(&intStack, &val);
        printf("This just popped: %d\n", val);
    }
    StackDispose(&intStack);
}
```

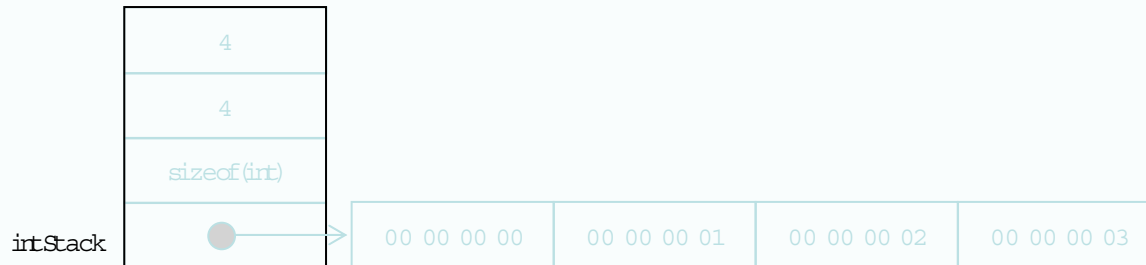


client.c

```
int main(int argc, char *argv[])
{
    int val;
    stack intStack;

    StackNew(&intStack, sizeof(int));
    for (val = 0; val < 6; val++)
        StackPush(&intStack, &val);

    while (!StackEmpty(&intStack)) {
        StackPop(&intStack, &val);
        printf("This just popped: %d\n", val);
    }
    StackDispose(&intStack);
}
```

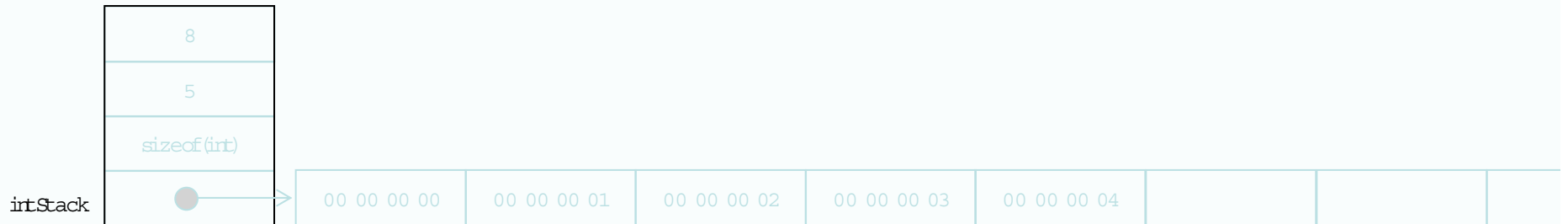


client.c

```
int main(int argc, char *argv[])
{
    int val;
    stack intStack;

    StackNew(&intStack, sizeof(int));
    for (val = 0; val < 6; val++)
        StackPush(&intStack, &val);

    while (!StackEmpty(&intStack)) {
        StackPop(&intStack, &val);
        printf("This just popped: %d\n", val);
    }
    StackDispose(&intStack);
}
```

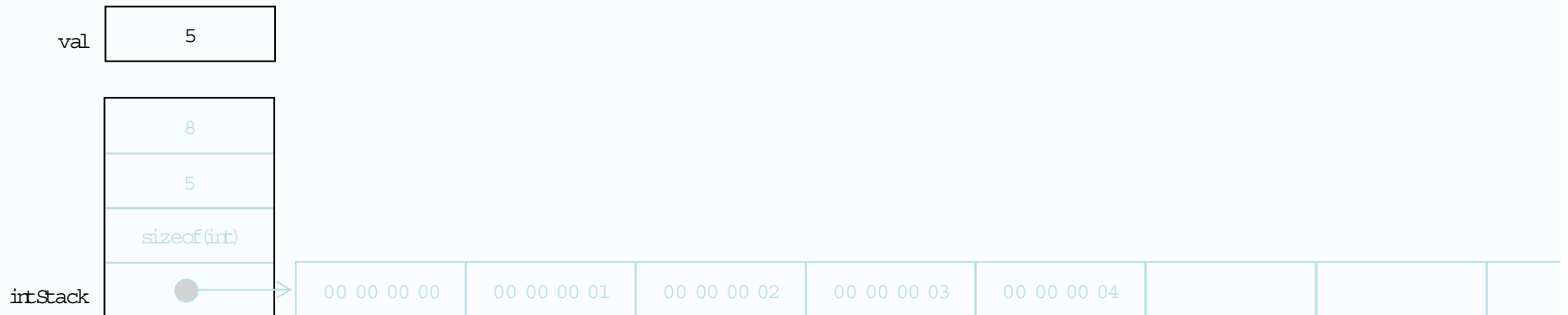


client.c

```
int main(int argc, char *argv[])
{
    int val;
    stack intStack;

    StackNew(&intStack, sizeof(int));
    for (val = 0; val < 6; val++)
        StackPush(&intStack, &val);

    while (!StackEmpty(&intStack)) {
        StackPop(&intStack, &val);
        printf("This just popped: %d\n", val);
    }
    StackDispose(&intStack);
}
```



client.c

```
int main(int argc, char *argv[])
{
    int val;
    stack intStack;

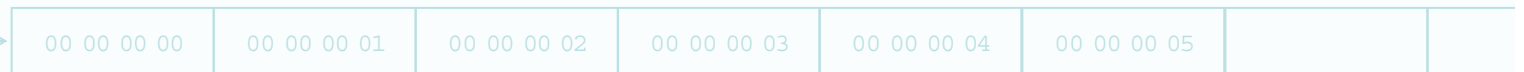
    StackNew(&intStack, sizeof(int));
    for (val = 0; val < 6; val++)
        StackPush(&intStack, &val);

    while (!StackEmpty(&intStack)) {
        StackPop(&intStack, &val);
        printf("This just popped: %d\n", val);
    }
    StackDispose(&intStack);
}
```

val



intStack



client.c

```
int main(int argc, char *argv[])
{
    int val;
    stack intStack;

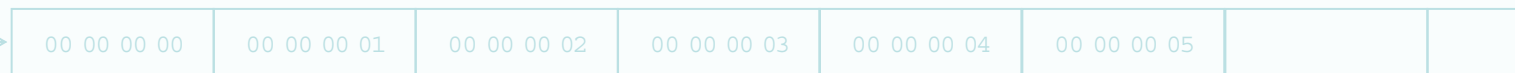
    StackNew(&intStack, sizeof(int));
    for (val = 0; val < 6; val++)
        StackPush(&intStack, &val);

    while (!StackEmpty(&intStack)) {
        StackPop(&intStack, &val);
        printf("This just popped: %d\n", val);
    }
    StackDispose(&intStack);
}
```

val



intStack



client.c

```
int main(int argc, char *argv[])
{
    int val;
    stack intStack;

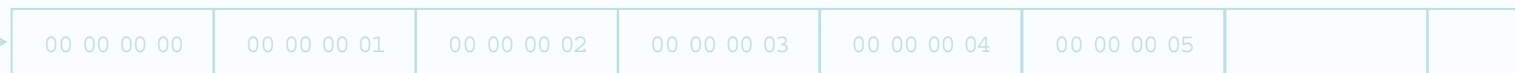
    StackNew(&intStack, sizeof(int));
    for (val = 0; val < 6; val++)
        StackPush(&intStack, &val);

    while (!StackEmpty(&intStack)) {
        StackPop(&intStack, &val);
        printf("This just popped: %d\n", val);
    }
    StackDispose(&intStack);
}
```

val



intStack

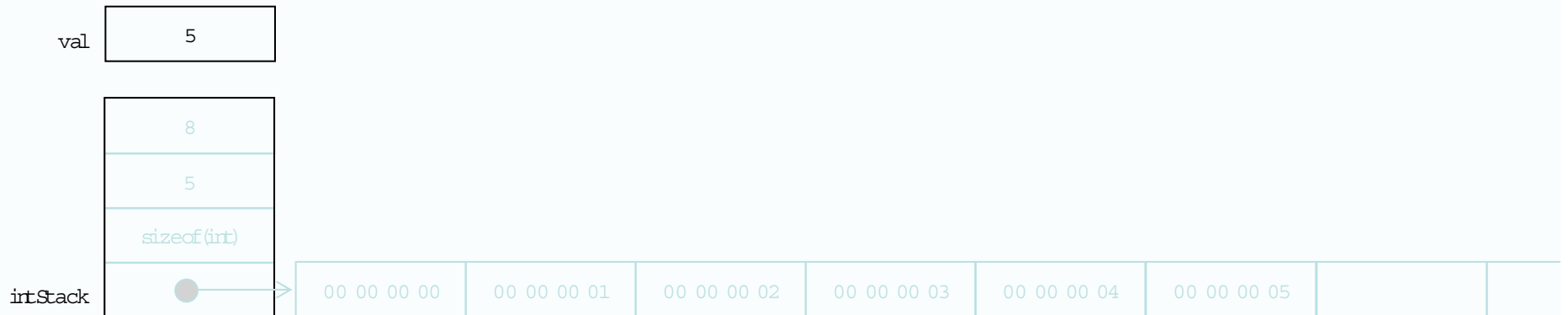


client.c

```
int main(int argc, char *argv[])
{
    int val;
    stack intStack;

    StackNew(&intStack, sizeof(int));
    for (val = 0; val < 6; val++)
        StackPush(&intStack, &val);

    while (!StackEmpty(&intStack)) {
        StackPop(&intStack, &val);
        printf("This just popped: %d\n", val);
    }
    StackDispose(&intStack);
}
```



client.c

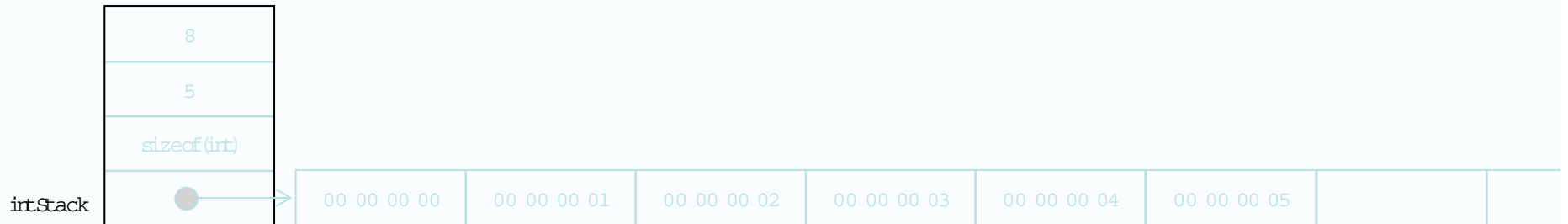
```
int main(int argc, char *argv[])
{
    int val;
    stack intStack;

    StackNew(&intStack, sizeof(int));
    for (val = 0; val < 6; val++)
        StackPush(&intStack, &val);

    while (!StackEmpty(&intStack)) {
        StackPop(&intStack, &val);
        printf("This just popped: %d\n", val);
    }
    StackDispose(&intStack);
}
```

val

5



Output

This just popped: 5

client.c

```
int main(int argc, char *argv[])
{
    int val;
    stack intStack;

    StackNew(&intStack, sizeof(int));
    for (val = 0; val < 6; val++)
        StackPush(&intStack, &val);

    while (!StackEmpty(&intStack)) {
        StackPop(&intStack, &val);
        printf("This just popped: %d\n", val);
    }
    StackDispose(&intStack);
}
```

val

5



Output

This just popped: 5

client.c

```
int main(int argc, char *argv[])
{
    int val;
    stack intStack;

    StackNew(&intStack, sizeof(int));
    for (val = 0; val < 6; val++)
        StackPush(&intStack, &val);

    while (!StackEmpty(&intStack)) {
        StackPop(&intStack, &val);
        printf("This just popped: %d\n", val);
    }
    StackDispose(&intStack);
}
```

val

4



Output

This just popped: 5

client.c

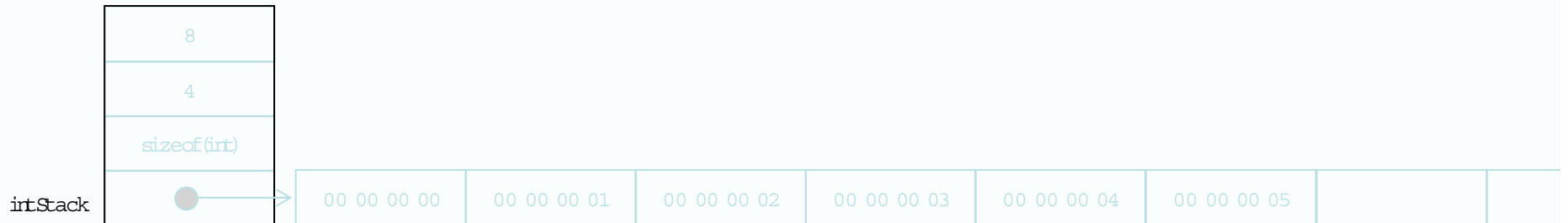
```
int main(int argc, char *argv[])
{
    int val;
    stack intStack;

    StackNew(&intStack, sizeof(int));
    for (val = 0; val < 6; val++)
        StackPush(&intStack, &val);

    while (!StackEmpty(&intStack)) {
        StackPop(&intStack, &val);
        printf("This just popped: %d\n", val);
    }
    StackDispose(&intStack);
}
```

val

4



Output

This just popped: 5

This just popped: 4

client.c

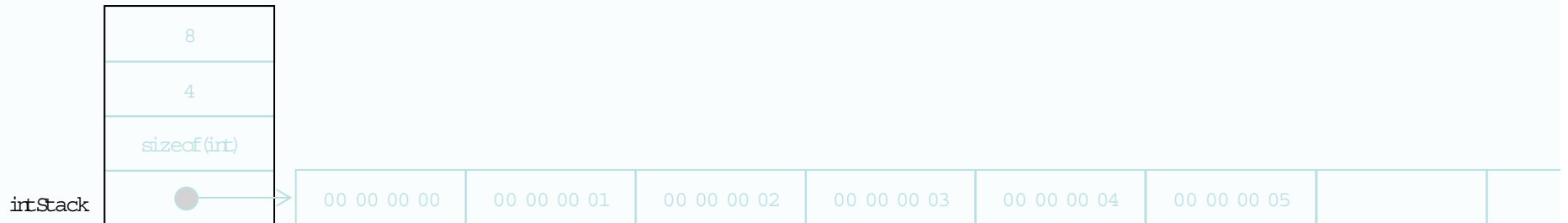
```
int main(int argc, char *argv[])
{
    int val;
    stack intStack;

    StackNew(&intStack, sizeof(int));
    for (val = 0; val < 6; val++)
        StackPush(&intStack, &val);

    while (!StackEmpty(&intStack)) {
        StackPop(&intStack, &val);
        printf("This just popped: %d\n", val);
    }
    StackDispose(&intStack);
}
```

val

4



Output

This just popped: 5

This just popped: 4

client.c

```
int main(int argc, char *argv[])
{
    int val;
    stack intStack;

    StackNew(&intStack, sizeof(int));
    for (val = 0; val < 6; val++)
        StackPush(&intStack, &val);

    while (!StackEmpty(&intStack)) {
        StackPop(&intStack, &val);
        printf("This just popped: %d\n", val);
    }
    StackDispose(&intStack);
}
```

val

3



Output

This just popped: 5

This just popped: 4

client.c

```
int main(int argc, char *argv[])
{
    int val;
    stack intStack;

    StackNew(&intStack, sizeof(int));
    for (val = 0; val < 6; val++)
        StackPush(&intStack, &val);

    while (!StackEmpty(&intStack)) {
        StackPop(&intStack, &val);
        printf("This just popped: %d\n", val);
    }
    StackDispose(&intStack);
}
```

val

3



Output

```
This just popped: 5
This just popped: 4
This just popped: 3
```

client.c

```
int main(int argc, char *argv[])
{
    int val;
    stack intStack;

    StackNew(&intStack, sizeof(int));
    for (val = 0; val < 6; val++)
        StackPush(&intStack, &val);

    while (!StackEmpty(&intStack)) {
        StackPop(&intStack, &val);
        printf("This just popped: %d\n", val);
    }
    StackDispose(&intStack);
}
```

val

3



Output

```
This just popped: 5
This just popped: 4
This just popped: 3
```


client.c

```
int main(int argc, char *argv[])
{
    int val;
    stack intStack;

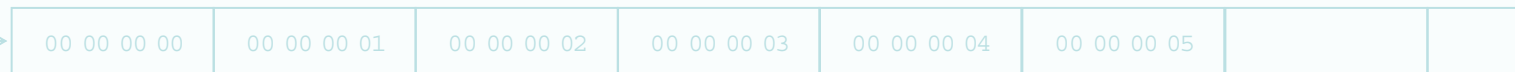
    StackNew(&intStack, sizeof(int));
    for (val = 0; val < 6; val++)
        StackPush(&intStack, &val);

    while (!StackEmpty(&intStack)) {
        StackPop(&intStack, &val);
        printf("This just popped: %d\n", val);
    }
    StackDispose(&intStack);
}
```

val



intStack



Output

This just popped: 5

This just popped: 4

This just popped: 3

client.c

```
int main(int argc, char *argv[])
{
    int val;
    stack intStack;

    StackNew(&intStack, sizeof(int));
    for (val = 0; val < 6; val++)
        StackPush(&intStack, &val);

    while (!StackEmpty(&intStack)) {
        StackPop(&intStack, &val);
        printf("This just popped: %d\n", val);
    }
    StackDispose(&intStack);
}
```

val

2



Output

```
This just popped: 5
This just popped: 4
This just popped: 3
This just popped: 2
```

client.c

```
int main(int argc, char *argv[])
{
    int val;
    stack intStack;

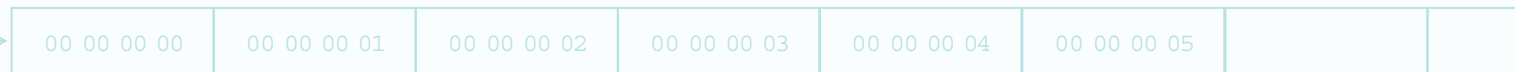
    StackNew(&intStack, sizeof(int));
    for (val = 0; val < 6; val++)
        StackPush(&intStack, &val);

    while (!StackEmpty(&intStack)) {
        StackPop(&intStack, &val);
        printf("This just popped: %d\n", val);
    }
    StackDispose(&intStack);
}
```

val



intStack



Output

```
This just popped: 5
This just popped: 4
This just popped: 3
This just popped: 2
```

client.c

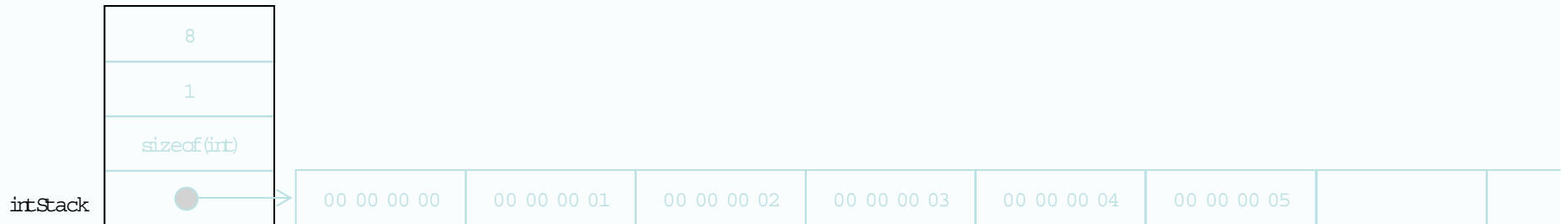
```
int main(int argc, char *argv[])
{
    int val;
    stack intStack;

    StackNew(&intStack, sizeof(int));
    for (val = 0; val < 6; val++)
        StackPush(&intStack, &val);

    while (!StackEmpty(&intStack)) {
        StackPop(&intStack, &val);
        printf("This just popped: %d\n", val);
    }
    StackDispose(&intStack);
}
```

val

1



Output

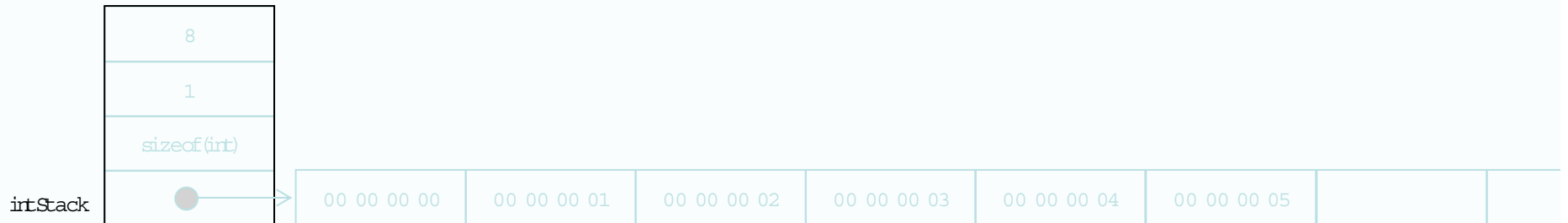
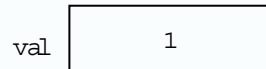
```
This just popped: 5
This just popped: 4
This just popped: 3
This just popped: 2
```

client.c

```
int main(int argc, char *argv[])
{
    int val;
    stack intStack;

    StackNew(&intStack, sizeof(int));
    for (val = 0; val < 6; val++)
        StackPush(&intStack, &val);

    while (!StackEmpty(&intStack)) {
        StackPop(&intStack, &val);
        printf("This just popped: %d\n", val);
    }
    StackDispose(&intStack);
}
```



Output

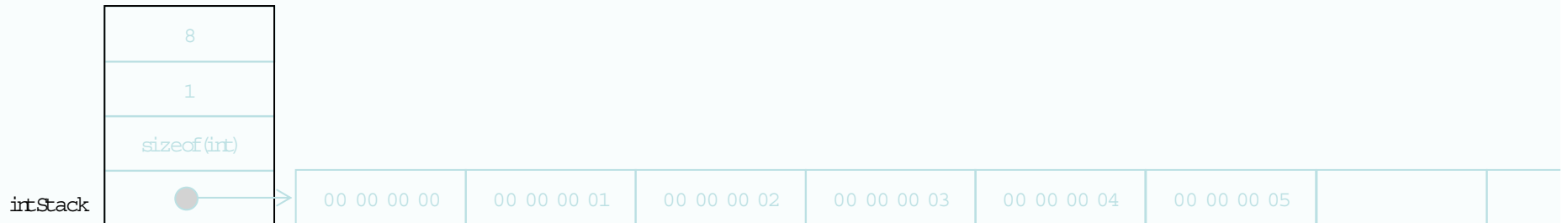
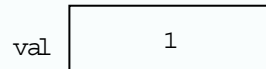
```
This just popped: 5
This just popped: 4
This just popped: 3
This just popped: 2
This just popped: 1
```

client.c

```
int main(int argc, char *argv[])
{
    int val;
    stack intStack;

    StackNew(&intStack, sizeof(int));
    for (val = 0; val < 6; val++)
        StackPush(&intStack, &val);

    while (!StackEmpty(&intStack)) {
        StackPop(&intStack, &val);
        printf("This just popped: %d\n", val);
    }
    StackDispose(&intStack);
}
```



Output

```
This just popped: 5  
This just popped: 4  
This just popped: 3  
This just popped: 2  
This just popped: 1
```

client.c

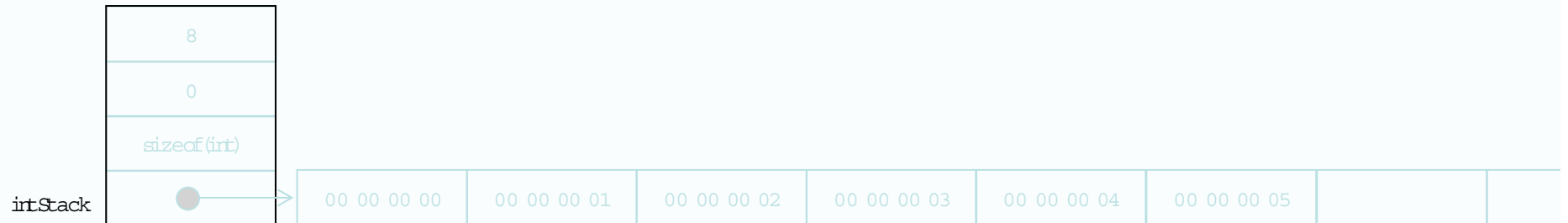
```
int main(int argc, char *argv[])
{
    int val;
    stack intStack;

    StackNew(&intStack, sizeof(int));
    for (val = 0; val < 6; val++)
        StackPush(&intStack, &val);

    while (!StackEmpty(&intStack)) {
        StackPop(&intStack, &val);
        printf("This just popped: %d\n", val);
    }
    StackDispose(&intStack);
}
```

val

0



Output

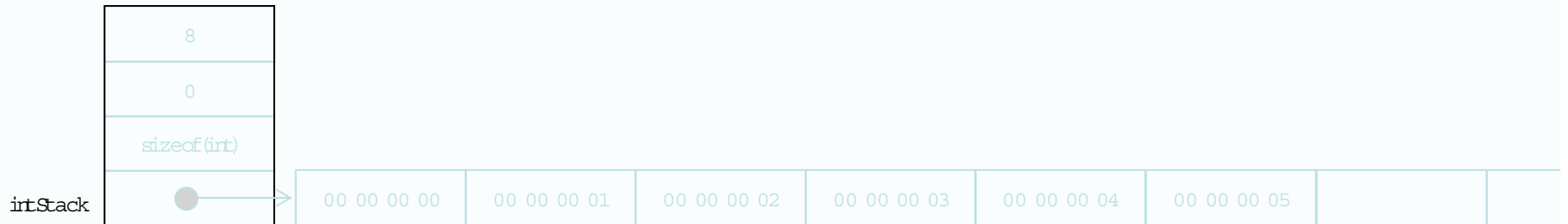
```
This just popped: 5
This just popped: 4
This just popped: 3
This just popped: 2
This just popped: 1
```

client.c

```
int main(int argc, char *argv[])
{
    int val;
    stack intStack;

    StackNew(&intStack, sizeof(int));
    for (val = 0; val < 6; val++)
        StackPush(&intStack, &val);

    while (!StackEmpty(&intStack)) {
        StackPop(&intStack, &val);
        printf("This just popped: %d\n", val);
    }
    StackDispose(&intStack);
}
```



Output

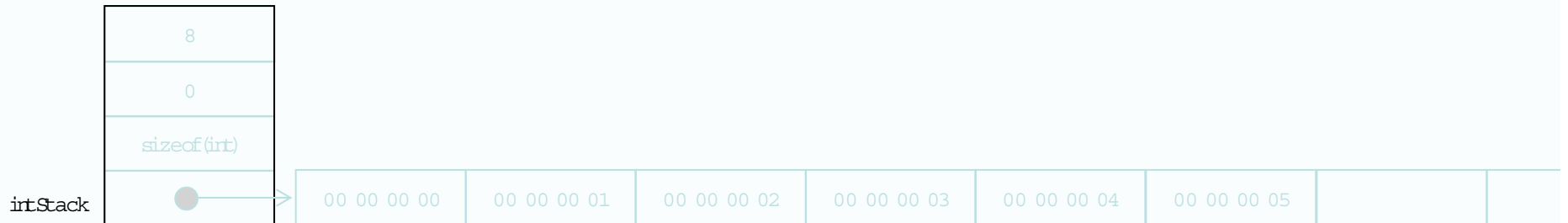
```
This just popped: 5
This just popped: 4
This just popped: 3
This just popped: 2
This just popped: 1
This just popped: 0
```


client.c

```
int main(int argc, char *argv[])
{
    int val;
    stack intStack;

    StackNew(&intStack, sizeof(int));
    for (val = 0; val < 6; val++)
        StackPush(&intStack, &val);

    while (!StackEmpty(&intStack)) {
        StackPop(&intStack, &val);
        printf("This just popped: %d\n", val);
    }
    StackDispose(&intStack);
}
```



Output

```
This just popped: 5
This just popped: 4
This just popped: 3
This just popped: 2
This just popped: 1
This just popped: 0
```

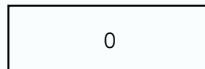
client.c

```
int main(int argc, char *argv[])
{
    int val;
    stack intStack;

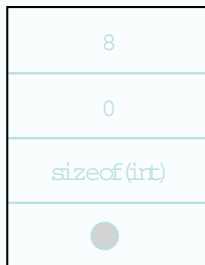
    StackNew(&intStack, sizeof(int));
    for (val = 0; val < 6; val++)
        StackPush(&intStack, &val);

    while (!StackEmpty(&intStack)) {
        StackPop(&intStack, &val);
        printf("This just popped: %d\n", val);
    }
    StackDispose(&intStack);
}
```

val



intStack



Output

```
This just popped: 5
This just popped: 4
This just popped: 3
This just popped: 2
This just popped: 1
This just popped: 0
```

client.c

```
int main(int argc, char *argv[])
{
    int val;
    stack intStack;

    StackNew(&intStack, sizeof(int));
    for (val = 0; val < 6; val++)
        StackPush(&intStack, &val);

    while (!StackEmpty(&intStack)) {
        StackPop(&intStack, &val);
        printf("This just popped: %d\n", val);
    }
    StackDispose(&intStack);
}
```

Output

```
This just popped: 5
This just popped: 4
This just popped: 3
This just popped: 2
This just popped: 1
This just popped: 0
```

stack.h

```
typedef struct {
    void *elems;
    int elemSize;
    int logLength;
    int allocLength;
} stack;

void StackNew(stack *s, int elemSize);
void StackDispose(stack *s);
bool StackEmpty(const stack *s);
void StackPush(stack *s, const void *elemAddr);
void StackPop(stack *s, void *elemAddr);
```

stack.c

We're about to implement the five functions. Keep in mind that the implementation is generic, and never has the full story about what it's storing. It's only because the client passes the base element size to the constructor that it's capable of cloning client data behind the scenes.

Here comes an implementation that works for any primitive type.

stack.h

```
typedef struct {
    void *elems;
    int elemSize;
    int logLength;
    int allocLength;
} stack;

void StackNew(stack *s, int elemSize);
void StackDispose (stack *s);
bool StackEmpty(const stack *s);
void StackPush(stack *s, const void *elemAddr);
void StackPop(stack *s, void *elemAddr);
```

stack.c

```
#define kInitialAllocationSize 4

void StackNew(stack *s, int elemSize)
{
    assert(elemSize > 0);
    s->elemSize = elemSize;
    s->logLength = 0;
    s->allocLength = kInitialAllocationSize;
    s->elems = malloc(kInitialAllocationSize * elemSize);
    assert(s->elems != NULL);
}
```

stack.h

```
typedef struct {  
    void *elems;  
    int elemSize;  
    int logLength;  
    int allocLength;  
} stack;
```

```
void StackNew(stack *s, int elemSize);  
void StackDispose(stack *s);  
bool StackEmpty(const stack *s);  
void StackPush(stack *s, const void *elemAddr);  
void StackPop(stack *s, void *elemAddr);
```

stack.c

```
#define kInitialAllocationSize 4  
  
void StackNew(stack *s, int elemSize)  
{  
    assert(elemSize > 0);  
    s->elemSize = elemSize;  
    s->logLength = 0;  
    s->allocLength = kInitialAllocationSize;  
    s->elems = malloc(kInitialAllocationSize * elemSize);  
    assert(s->elems != NULL);  
}
```

Key observations:

- *The constructor requires that the client element size be identified up front. Since it doesn't (and will never) know the true data type, it needs the size so it at least knows how many bytes to replicate behind the scenes with every call to StackPush.*
- *The implementation chooses to allocate space for 4 client elements. The 4 is arbitrary, but it should probably be small, since a good number of applications never deal with stacks that are all that deep. If during the course of a program the stack saturates allocated memory, the implementation can just reallocate the storage using a (popular) doubling strategy. You'll see.*

stack.h

```
typedef struct {
    void *elems;
    int elemSize;
    int logLength;
    int allocLength;
} stack;

void StackNew(stack *s, int elemSize);
void StackDispose(stack *s);
bool StackEmpty(const stack *s);
void StackPush(stack *s, const void *elemAddr);
void StackPop(stack *s, void *elemAddr);
```

stack.c

```
#define kInitialAllocationSize 4

void StackDispose (stack *s)
{
    free(s->elems);
}
```

Key observations:

- *The destructor simply needs to dispose of any resources that'll otherwise be orphaned. The client has no clue that one single block of dynamically allocated memory is being used to house all of the elements, so the implementation needs to properly free that memory, else it'll be orphaned and unavailable for the lifetime of the application.*
- *Note that we don't call free(s). StackNew doesn't allocate space for the stack struct—the client does! The stack destructor should only clean up its own mess, not someone else's.*

stack.h

```
typedef struct {
    void *elems;
    int elemSize;
    int logLength;
    int allocLength;
} stack;

void StackNew(stack *s, int elemSize);
void StackDispose(stack *s);
bool StackEmpty(const stack *s);
void StackPush(stack *s, const void *elemAddr);
void StackPop(stack *s, void *elemAddr);
```

stack.c

```
#define kInitialAllocationSize 4

void StackDispose (stack *s)
{
    free(s->elems);
}

bool StackEmpty(const stack *s)
{
    return s->logLength == 0;
}
```

Key observations:

- *The destructor simply needs to dispose of any resources that'll otherwise be orphaned. The client has no clue that one single block of dynamically allocated memory is being used to house all of the elements, so the implementation needs to properly free that memory, else it'll be orphaned and unavailable for the lifetime of the application.*
- *Note that we don't call free(s). StackNew doesn't allocate space for the stack struct—the client does! The stack destructor should only clean up its own mess, not someone else's.*
- *StackEmpty is a no-brainer.*

stack.h

```
typedef struct {
    void *elems;
    int elemSize;
    int logLength;
    int allocLength;
} stack;

void StackNew(stack *s, int elemSize);
void StackDispose(stack *s);
bool StackEmpty(const stack *s);
void StackPush(stack *s, const void *elemAddr);
void StackPop(stack *s, void *elemAddr);
```

stack.c

```
void StackPush(stack *s, const void *elemAddr)
{
    void *destAddr;
    if (s->logLength == s->allocLength) {
        s->allocLength *= 2;
        s->elems = realloc(s->elems, s->allocLength * s->elemSize);
        assert(s->elems != NULL);
    }

    destAddr = (char *)s->elems + s->logLength * s->elemSize;
    memcpy(destAddr, elemAddr, s->elemSize);
    s->logLength++;
}
```

stack.h

```
typedef struct {
    void *elems;
    int elemSize;
    int logLength;
    int allocLength;
} stack;

void StackNew(stack *s, int elemSize);
void StackDispose(stack *s);
bool StackEmpty(const stack *s);
void StackPush(stack *s, const void *elemAddr);
void StackPop(stack *s, void *elemAddr);
```

stack.c

```
void StackPush(stack *s, const void *elemAddr)
{
    void *destAddr;
    if (s->logLength == s->allocLength) {
        s->allocLength *= 2;
        s->elems = realloc(s->elems, s->allocLength * s->elemSize);
        assert(s->elems != NULL);
    }

    destAddr = (char *)s->elems + s->logLength * s->elemSize;
    memcpy(destAddr, elemAddr, s->elemSize);
    s->logLength++;
}
```

The devil's in the details:

- *The last three lines replicate the byte pattern addressed by elemAddr, placing it at the manually computed address within raw storage.*
- *Note that the copy is a shallow copy. If the s->elemSize bytes copied in are pointers to dynamically allocated memory, then ownership of that memory is transferred from the client to the implementation.*

stack.h

```
typedef struct {
    void *elems;
    int elemSize;
    int logLength;
    int allocLength;
} stack;

void StackNew(stack *s, int elemSize);
void StackDispose(stack *s);
bool StackEmpty(const stack *s);
void StackPush(stack *s, const void *elemAddr);
void StackPop(stack *s, void *elemAddr);
```

stack.c

```
void StackPush(stack *s, const void *elemAddr)
{
    void *destAddr;
    if (s->logLength == s->allocLength) {
        s->allocLength *= 2;
        s->elems = realloc(s->elems, s->allocLength * s->elemSize);
        assert(s->elems != NULL);
    }

    destAddr = (char *)s->elems + s->logLength * s->elemSize;
    memcpy(destAddr, elemAddr, s->elemSize);
    s->logLength++;
}
```

The devil's in the details:

- *The last three lines replicate the byte pattern addressed by elemAddr, placing it at the manually computed address within raw storage.*
- *Note that the copy is a shallow copy. If the s->elemSize bytes copied in are pointers to dynamically allocated memory, then ownership of that memory is transferred from the client to the implementation.*
- *The conditional reallocation only applies if we've previously saturated memory. Type 'man realloc' at the command line to get the full documentation on how realloc works.*

stack.h

```
typedef struct {
    void *elems;
    int elemSize;
    int logLength;
    int allocLength;
} stack;

void StackNew(stack *s, int elemSize);
void StackDispose(stack *s);
bool StackEmpty(const stack *s);
void StackPush(stack *s, const void *elemAddr);
void StackPop(stack *s, void *elemAddr);
```

stack.c

```
void StackPop(stack *s, void *elemAddr)
{
    const void *sourceAddr;

    assert(!StackEmpty(s));
    s->logLength--;
    sourceAddr = (const char *) s->elems + s->logLength * s->elemSize;
    memcpy(elemAddr, sourceAddr, s->elemSize);
}
```

Final words on implementation

- *StackPop is, not surprisingly, the reverse of StackPush. It needs to transfer ownership of the byte pattern of the topmost element back to the client, and it does so with a symmetric call to memcpy. We manually compute the address of the byte pattern in precisely the same way, and trust (how can we not?) that the client has supplied the address of a figure large enough to accommodate the byte transfer.*
- *We never reallocate the array to be smaller, even if we dip below some 50% saturation value. In practice, realloc punts on any request to shrink the array anyway, so it's typically a wasted effort.*