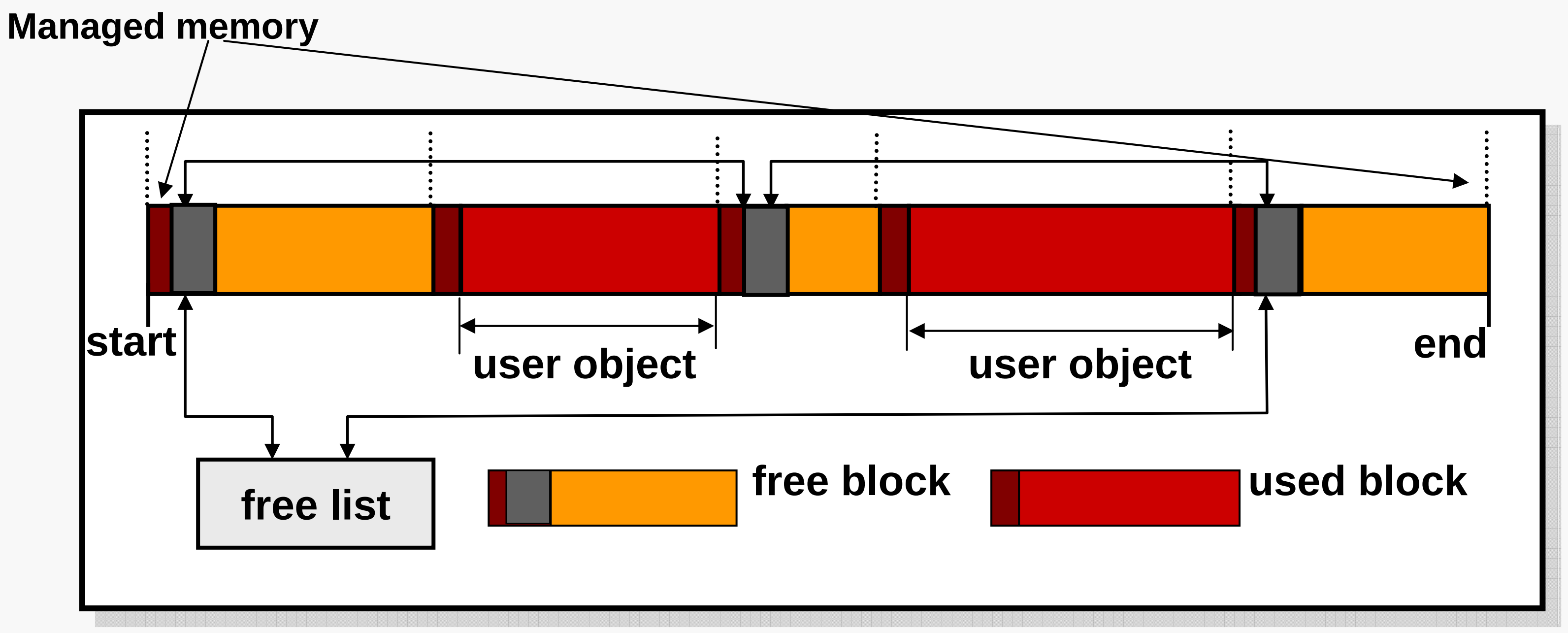
Project 0: User-level Memory Allocator

The memory resource consists of a collection of blocks, some allocated to users and some free to be allocated in the future.

Memory is just a sequence of bytes, each with a unique address (offset).

Allocator must keep track of where free and used blocks are.

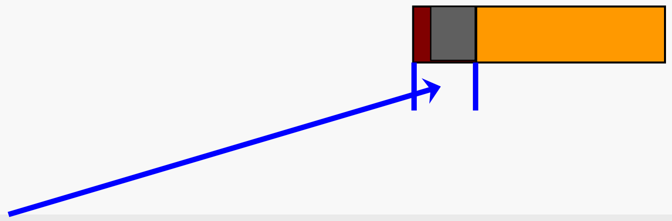


Project 0: Data Structures

It suits our current purpose to use a simple list (supplied in the pintos source code) to track the used and free blocks.

Remember that the list nodes must also be allocated from the managed memory.

What information must be maintained for a free block? Is there any difference for a used block?

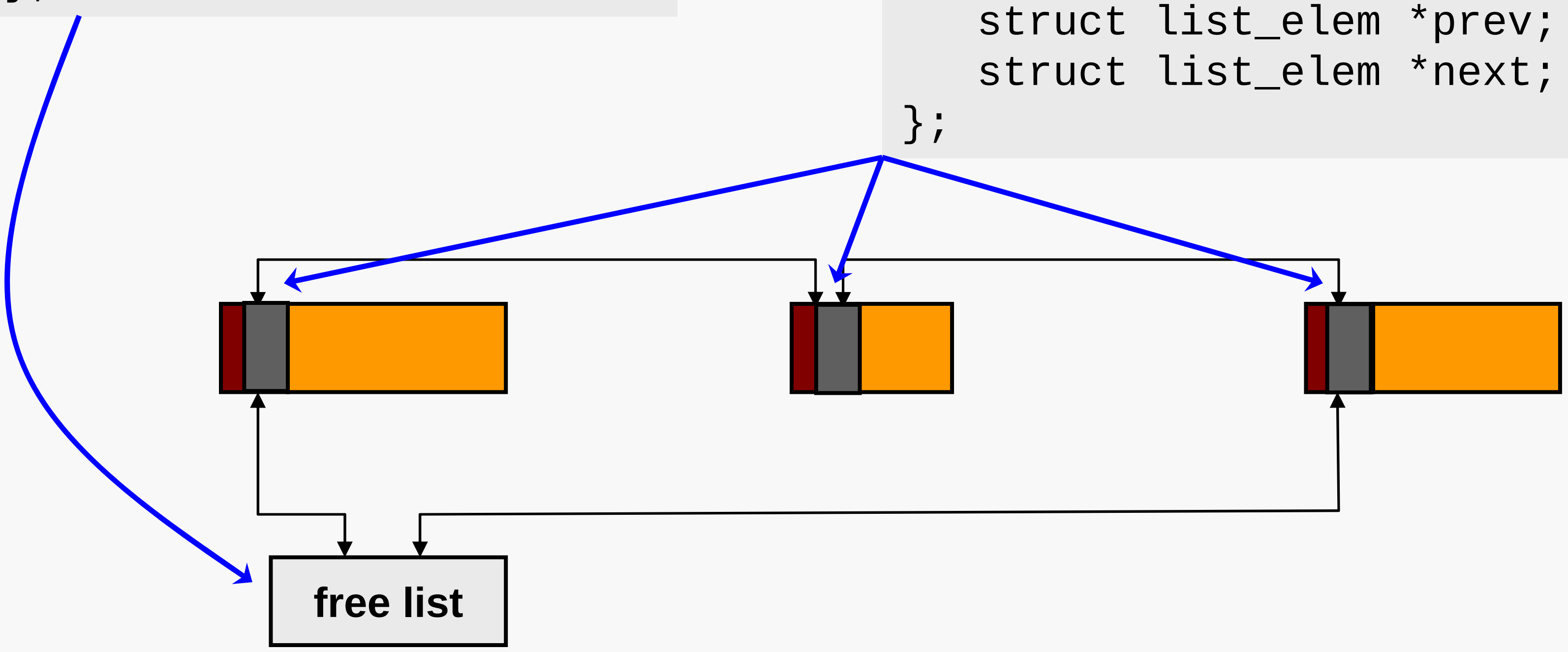


```
struct free_block {
   size_t           length; /* length of block,
                               including header */
   struct list_elem elem;   /* list element for free list */
};
```

Project 0: pintos `list`

```
struct list {
   struct list_elem head;
   struct list_elem tail;
};
```

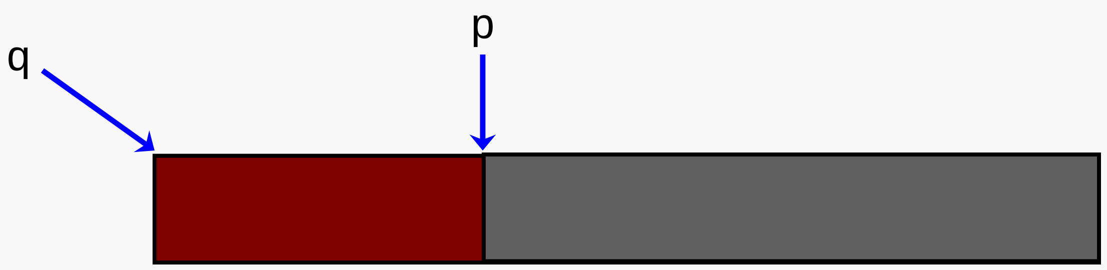
```
struct list_elem {
   struct list_elem *prev;
   struct list_elem *next;
};
```



Project 0: address computations

Problem:

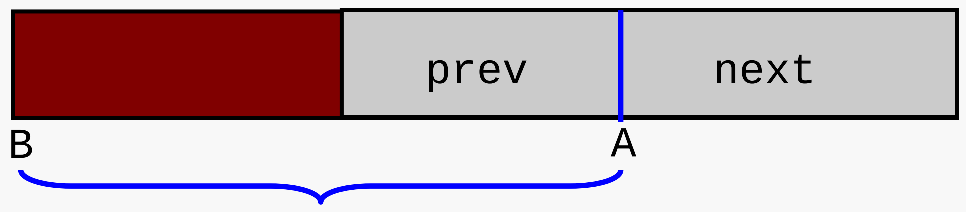
- we have a pintos `list` pointer `p`, which targets a `list_elem` object
- we need a pointer `q` to the surrounding `free_block` object



We don't want to make any assumptions about the layout in memory of the `free-block` object that could cause problems if the object specification changed.

Project 0: address computations

If we knew one fact, the address we need would be easy to compute:



`C`: offset of `next` within the `free_block` object

We know the address A.

If we knew `C`, then the address of the `free_block` object would seem to be:

```
B = A - C
```

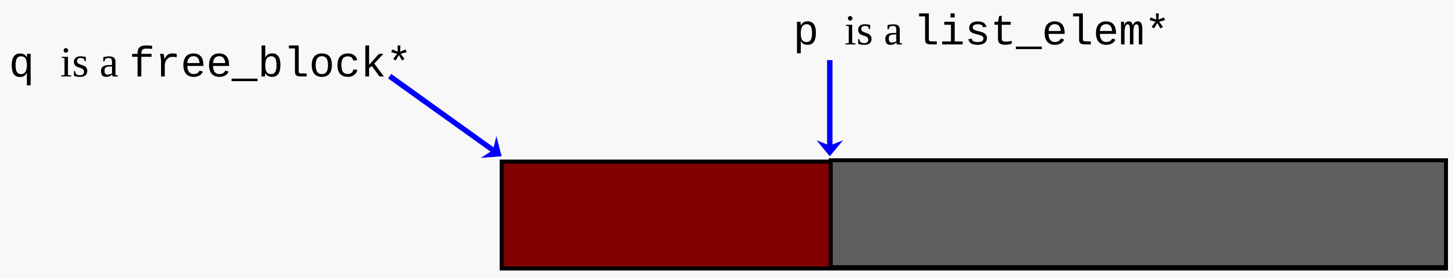
But the semantics of pointer arithmetic get in the way…

Project 0: pointer arithmetic

From the C Standard:

When two pointers are subtracted, both shall point to elements of the same array object, or one past the last element of the array object; the result is the difference of the subscripts of the two array elements. The size of the result is implementation-defined, and its type (a signed integer type) is **ptrdiff_t** defined in the **<stddef.h>** header. If the result is not representable in an object of that type, the behavior is undefined. In other words, if the expressions **P** and **Q** point to, respectively, the *i*-th and *j*-th elements of an array object, the expression **(P)-(Q)** has the value $i-j$ provided the value fits in an object of type **ptrdiff_t**.

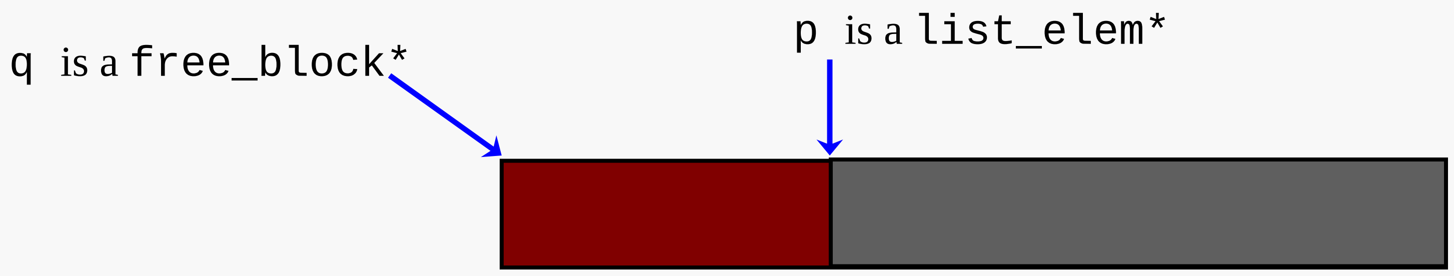
We have type issues.



Project 0: pointer casts

We need to cast the relevant pointer types so that the computation yields the address of the 0-th byte of the `free_block` object:

```
q = (free_block*)((uint8_t*)(&p->next) – C)
```



But how do we get the offset of a member of an object?

Project 0: `offsetof()`

The C Standard Library provides a solution:

offsetof(*type***,** *member-designator***)**

expands to an integer constant expression that has type **size_t**, the value of which is the offset in bytes, to the structure member (designated by *member-designator*), from the beginning of its structure (designated by *type*).

So we could write something like:

```
q = (free_block*)((uint8_t*)(&p->next) –
             offsetof(struct free_block, elem.next))
```

Project 0: pintos `list_entry()` macro

Since this sort of thing comes up quite a bit when pintos `list` objects are used, it's in the spirit of C to create a generic macro that the pre-processor can expand:

```
#define list_entry(LIST_ELEM, STRUCT, MEMBER)           \
        ((STRUCT *) ((uint8_t *) &(LIST_ELEM)->next     \
                    - offsetof (STRUCT, MEMBER.next)))
```

Then if we write:

```
free_block *q = list_entry(p, struct free_block, elem);
```

the pre-processor turns that into:

```
free_block *q = (struct free_block *)
                  ((unit8_t*)&p->next –
                   offsetof(struct free_block, elem.next)));
```

Project 0: concurrency issues

Remember the purpose of this assignment is to implement (or at least simulate) the operation of a very simple memory allocator in an operating system.

Modern OS designs provide for multitasking, and perhaps even true concurrency.

That means that the implementation of the memory manager must be "thread-safe".

That is, it is entirely possible that a user process P may call `mem_alloc()` to request memory, but that P may be temporarily suspended and another user process Q allowed to run before P's call to `mem_alloc()` has completed.

Consider the implications…

What if the interruption occurs while `mem_alloc()` is choosing a free block to use? or during the execution of one of the `list` functions called by `mem_alloc()`?

Project 0: critical sections

A *critical section* is a code segment in which a process is accessing, especially modifying, shared data.

For example, consider what might happen if we have two or more processes in the following situation:

```
static int counter = 0;  // a global variable to protect

// function executed by each thread
void increment() {
    int i;
    for (i = 0; i < 1000000; i++) {
        // critical section begins
        counter++;
        // critical section ends
    }
}
```

Project 0: mutual exclusion

What we need is a way to specify, in code, that only one process is allowed to enter a block of code at any given time:

```
// function executed by each thread
void increment() {
    int i;
    for (i = 0; i < 1000000; i++) {
        set_lock();    // atomic operation
        counter++;
        unset_lock();
    }
}
```

Alas, the C Standard Library doesn't provide such functions.

Project 0: `pthread` mutexes

The pthread (Posix thread) library provides one solution:

```
// global declaration:
static pthread_mutex_t lock = PTHREAD_MUTEX_INITIALIZER;
```

The user code now must set and unset the lock at the appropriate places:

```
// function executed by each thread
void increment() {
    int i;
    for (i = 0; i < 1000000; i++) {
        pthread_mutex_lock( &lock );
        counter++;
        pthread_mutex_unlock( &lock );
    }
}
```

Project 0: general comments

Read carefully before you write too much code:

- `list.h` and `list.c`
- `memalloc.h`
- `test_mem.c`

You might want to comment out some of the tests at first, and perhaps even modify the given test code to get a feel for what's going on.

We will test your solution with the original code though.

Most of your work will go into writing `memalloc.c` to implement the functions declared in `memalloc.h`.

The given test code shows some examples of how to use the pthread library to spawn off multiple threads executing the same function; interesting but not relevant to your own code.