[redacted]

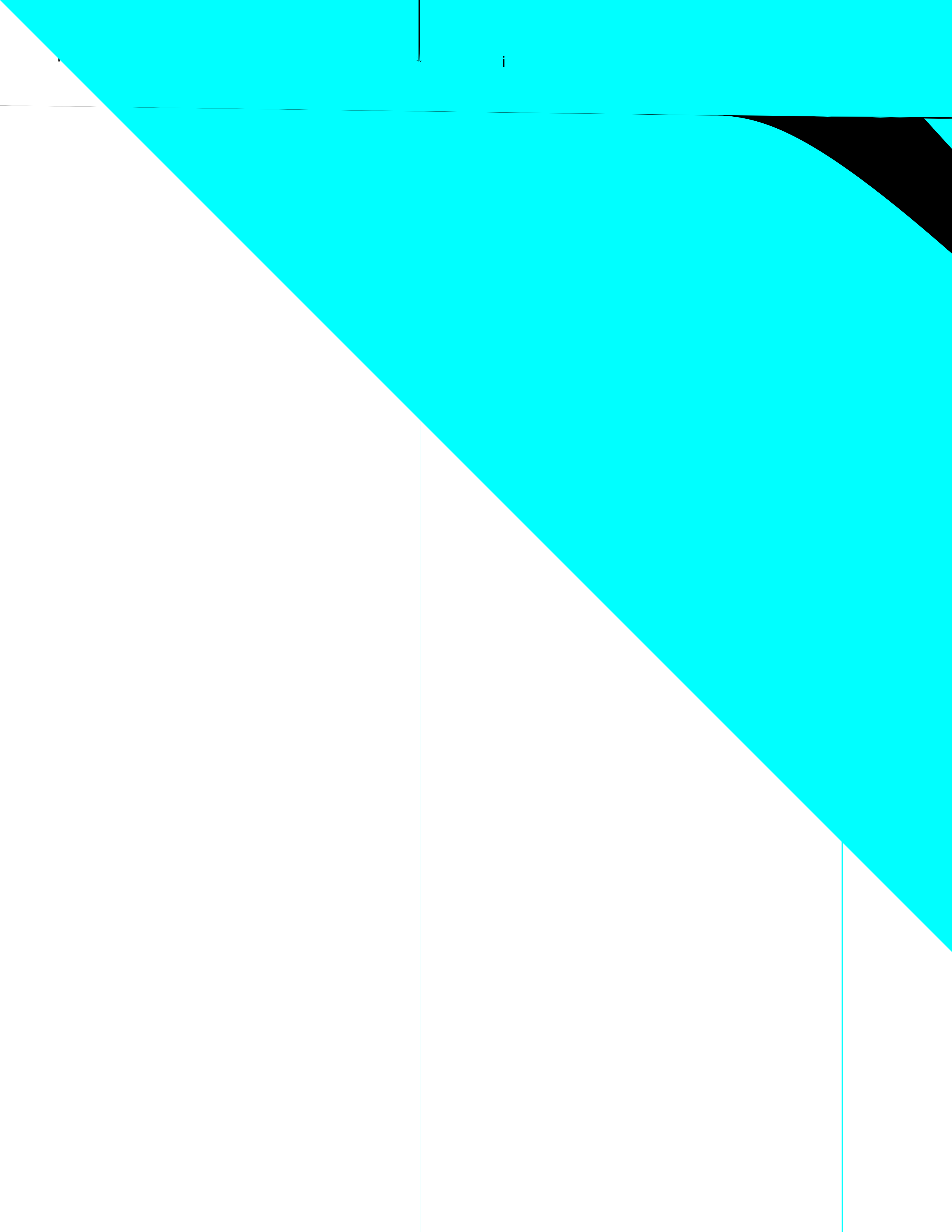
rameter `T` which will represent the type that is stored in our vector. This will be templatized at compile-time, similar to how `vector<T>` is in C++.

The `data` field is a [flexible array member](#) from C99.

Note: We will forgo error checking of `malloc` and `realloc` for simplicity.

`new`

The `new` function should `malloc` enough memory for some initial members. The size of the required storage



```
#define qvec_push(v, i)                                        \
({                                                             \
    if (v->len >= v->cap) {                                    \
        v->cap *= 2;                                           \
        v = realloc(v, sizeof(?) + v->cap * sizeof(?)); \
    }                                                          \
    v->data[v->len++] = (i);                                   \
})
```

we might be left wondering what to insert into the `?` marked locations.

The second `?` is less worrying. This should be `sizeof(T)`. We could just pass the type again, but doing it on every push is not ideal. In fact, we don't need any new information. Recall that the `data` field of `qvec` is of type `T[]`. Performing a dereference of this will give us the size of a single `T`, exactly what we want!

The first `?` is more bothersome. We are interested in determining the value of `sizeof(qvec(T))`. We can't use the `data` field here, since the `T` required here is the actual typename used during initialization. This would be viable if it were possible to generate a type name from an arbitrary variable but unfortun

[redacted]

```
[redacted]
    struct qvec_base,[redacted]
    T data[];
[redacted]
```

[redacted]ows embedding of existing struct defi[redacted]

[redacted]nally, we can define our `push` function as:

```
[redacted]ne qvec_push(v, i)
[redacted]
[redacted]f (v->len >= v->cap) {
    v->cap *= 2;
    v = realloc(v, sizeof(qvec_base) + v->cap * sizeo[redacted]
[redacted]
[redacted]->data[v->len++] = (i);
[redacted]
```

`free`[redacted]

[redacted]we only use a single `malloc` to initialize the type, this is simply

```
[redacted]ne qvec_free(v) free(v)
```

API [redacted]o far

[redacted]ee what this gives us so far

```
[redacted](int) *iv = qvec_new(int);
[redacted]_push(iv, 5);
[redacted]_push(iv, 8);
[redacted]tf("%d\n", qvec_at(iv, 0));
[redacted]_at(iv, 1) = 5;
[redacted]_free(iv);
```

[redacted]ompared similar C++ vector usage

```
[redacted]:vector<int> iv;
[redacted]ush_back(5);
[redacted]ush_back(8);
[redacted]tf("%d\n", iv[0]);
[redacted]] = 5;
```

[redacted]

Looking okay, but lets go a bit further.

Extended Functions

Generic Printing

It is fairly common that we want to dump the values of a vector to see what is inside. If we wanted to write this for an integer vector, the following would work

```
#define qve
```

ed use case. Fortunately for us, C11 brings some new interesting features to the table which we can use.

The C11 `_Generic` keyword allows rudimentary switching based on the type of its input. Think of it just as a **compile-time switch statement on types**.

For example, we could construct a macro to print the name of a type

```
#define type_name(x) _Generic((x), int: "int", float: "float")
```

```
p
```

[redacted]

```
    for (int i = 0; i < v->len; ++i) {  \
        printf(GET_FMT_SPEC(v->data[i]), v->data[i]);\
        if (i + 1 < v->len)             \
            printf(", ");               \
    }                                   \
    printf("]\n");                      \
})
```

This would now work on an integer and float `qvec` type with no modifications. Of course, we could whatever types we need.

You may recall that I mentioned that we could solve an earlier issue regarding our `push` function if name from a variable. It seems like the `_Generic` keyword would help is achieve this and indeed i is that it is evaluated after preprocessing, so we cannot use its output as part of the preprocessor.

```
qvec(int) v = qvec_new(int, 1, 5, 2, 3);
```

`cleanup`

`void cleanup(T**)` `T`

`qvec`

Note that an attribute doesn't strictly need to be specified after the type definition.

This is nice, but if you had actually compiled the above you would get a numbe[redacted]

```
auto iv = qvec_new(int);
```

Although yet again, our expectations differ to reality. This will not compile! The reason for this is that previously we were relying on the inline struct definition of `qvec(T)` that was declared on every initialization. Without this declaration, our new `auto` keyword cannot find any struct which matches the return type and must fail.

As an example, the following works fine

```
qvec(int) *a = qvec_new(int);
auto b = qvec_new(int);
```

because the `qvec(int)` declared the struct, so the next `qvec` return type can be deduced correctly. This is simply an inherent limitation with the tools we have. A simple solution would be simply forward declare our structs.

```
qvec(int);

int main(void)
{
    auto a = qvec_new(int); // Ok!
}
```

But this is one extra line to type for each `qvec` type required!

Drawbacks

We have a pretty good set of functions associated with our `qvec` so far. Usability is ok and we have a few of the more desirable features of C++ in our hands within C.

Undoubtedly however, there are some inherent problems that we just can't solve.

Complex Container Types

We can do the following in C++

```
std::vector<std::vector<std::vector<int>>> v;
```

To do this with our `qvec` the following is required

```
typedef qvec(int) qvec_int;
typedef qvec(qvec_int) qvec_qvec_int;
qvec(qvec_qvec_int) *v = qvec_new(qvec_qvec_int);
```

Recall back to our `new` implementation. We generate a struct with a name `qvec_##T` where `T` is the type. Since this is concatenated to make an identifier, the types *must* be comprised only of characters which can exist within an identifier (`[_0-9A-Za-z]`). Any types which use other characters, such as functions, pointers and even our own `qvec` types must have a typedef before we can use them.

As an example, the following

```
qvec(char**);
```

khash