Ternary search tries have the following invariants:

- The left and right of a trie node act as a BST. In other words, for any tnode T, IGNORING middle pointers, every tnode in the T->left subtree represents a character whose ASCII value is less than T->c, and every tnode in the T->right subtree represents a character whose ASCII value is greater than T->c.
- For any tnode T, T->middle is a valid tnode.
- If, for some tnode T, T->middle == NULL, then T->elem must not be NULL (otherwise this trie would not be a prefix for any element!).

The same tnode_lookup function in the implementation can be used to implement both trie_member (the given string is in the trie) and trie_prefix (this string is a proper prefix of a string in the trie).

```c
1 tnode *tnode_lookup(tnode *T, char *s, size_t i) {
2     REQUIRES(is_tnode_root(T) && s != NULL && i < strlen(s));
3
4 }
5
6 bool trie_member(trie TR, char *s) {
7     REQUIRES(is_trie(TR) && s != NULL && strlen(s) > 0);
8     tnode *T = tnode_lookup(TR->root, s, 0);
9     return T != NULL && T->is_end;
10 }
11
12 bool trie_prefix(trie TR, char *s) {
13     REQUIRES(is_trie(TR) && s != NULL && strlen(s) > 0);
14     tnode *T = tnode_lookup(TR->root, s, 0);
15     return T != NULL && T->middle != NULL;
16 }
```
**Definition-as-use**

In C0, all variable declarations were comprised of a type and a variable name.

the type (int pointer) the variable name (myptr)

```c
int* myptr;
```

In C, you have to think about declarations differently: as a base type and a pattern describing how the variable name is used.

the type (int) it is used by deferencing (myptr)

```c
int *myptr;
```

On one level, this is just a stylistic difference: C isn’t that whitespace sensitive. But there are two places where this makes a critical difference.

**Checkpoint 0**

One one line, declare an integer i, a pointer to an integer p, and an array of pointers to integers A.

```c
int ____________________________;
```

**Checkpoint 1**

You can define and use a function pointer like this:

```c
1 typedef int string_hasher(char *s);
2 int hash_string(char *s) { REQUIRES(s != NULL); ... }
3 int main() {
4   string_hasher *f = &hash_string;
5   printf("Hash of \"hello world\": %d\n", (*f)("hello world");
6 }
```

Using the idea of definition-as-use, declare f in the example above without using the intermediate typedef.

```c
______________________________ = &hash_string;
```

**Checkpoint 2**

Consider the following:

```c
1 typedef int compare(void *, void *);
2 void sort(void **A, int len, compare *F);
3 // requires A != NULL
4 // requires \length(A) == len
5 // requires compare != NULL
```

How would you use this procedure to sort an array of strings? How would you write a generic implementation of sort using the selection sort algorithm from earlier in the semester? Which of these preconditions could we actually check in the sort function, and how?