const

Sometimes we use **const** with pointer parameters to indicate that the function will not / should not change what it points to. The actual pointer can be changed, however.

```
// This function promises to not change str's characters
int countUppercase(const char *str) {
    int count = 0;
    for (int i = 0; i < strlen(str); i++) {
        if (isupper(str[i])) {
            count++;
        }
    }
    return count;
}
```

const

By definition, C gets upset when you set a **non-const** pointer equal to a **const** pointer. You need to be consistent with **const** to reflect what you cannot modify.

```
// This function promises to not change str's characters
int countUppercase(const char *str) {
      // compiler warning and error
     char *strToModify = str;
     strToModify[0] = …
}
```

const

const can be confusing to interpret in some variable types.

```
// cannot modify this char
const char c = 'h';

// cannot modify chars pointed to by str
const char *str = …

// cannot modify chars pointed to by *strPtr
const char **strPtr = …
```

Structs

A **`struct`** is a way to define a new variable type that is a group of other variables.

```
struct date {           // declaring a struct type
    int month;
    int day;            // members of each date structure
};
…

struct date today;                      // construct structure instances
today.month = 1;
today.day = 28;

struct date new_years_eve = {12, 31};  // shorter initializer syntax
```

Structs

Wrap the struct definition in a **`typedef`** to avoid having to include the word **`struct`** every time you make a new variable of that type.

```
typedef struct date {
    int month;
    int day;
} date;
…

date today;
today.month = 1;
today.day = 28;

date new_years_eve = {12, 31};
```

Structs

If you pass a struct as a parameter, like for other parameters, C passes a **copy** of the entire struct. **Use a pointer to modify a specific instance.**

```
void advance_day(date *d) {
      (*d).day++;
}

int main(int argc, char *argv[]) {
      date my_date = {1, 28};
      advance_day(&my_date);
      printf("%d", my_date.day); // 29
      return 0;
}
```

Structs

The **arrow** operator lets you access the field of a struct pointed to by a pointer.

```
void advance_day(date *d) {
      d->day++;         // equivalent to (*d).day++;
}

int main(int argc, char *argv[]) {
      date my_date = {1, 28};
      advance_day(&my_date);
      printf("%d", my_date.day); // 29
      return 0;
}
```

Structs

`sizeof` gives you the entire size of a struct, which is the sum of the sizes of all its contents.

```
typedef struct date {
      int month;
      int day;
  } date;


int main(int argc, char *argv[]) {
      int size = sizeof(date);    // 8
      return 0;
}
```

Arrays of Structs

To initialize an entry of the array, you must use this special syntax to confirm the type to C.

```
typedef struct my_struct {
     int x;
     char c;
} my_struct;
```

…

```
my_struct array_of_structs[5];
array_of_structs[0] = (my_struct){0, 'A'};
```

Arrays of Structs

You can also set each field individually.

```
typedef struct my_struct {
     int x;
     char c;
} my_struct;

…
my_struct array_of_structs[5];
array_of_structs[0].x = 2;
array_of_structs[0].c = 'A';
```

Stacks

- A **Stack** is a data structure representing a stack of things.
- Objects can be ***pushed*** on top of or ***popped*** from the top of the stack.
- Only the top of the stack can be accessed; no other objects in the stack are visible.
- Main operations:
  - **`push(value)`**: add an element to the top of the stack
  - **`pop()`**: remove and return the top element in the stack
  - **`peek()`**: return (but do not remove) the top element in the stack