Structures

Structures

- A C structure is a collection of one or more variables of the same or different types
- Structures permit convenient handling of complicated data as a single unit
- Similar to records in Pascal
- Copying, assigning to, recovering address using & and accessing members are all legal operations on structures

Creating a structure

- The `struct` keyword is used to create structures

```
struct address {
  char * street;
  char * city;
  int zip;
} a, b;
```

- The tag `address` is optional but useful for identifying the struct so new variables can be created

```
struct address a, b;
```

Initializing structures

- Structures can be initialized just like arrays

```
struct address a = {"street", "recife",
  123456}, b;
```

- An automatic structure can also be initialized by assignment or by calling a function that returns the structure of the right type

```
a.zip = 123456;
a.street = "street";
a.city = "recife";
```

Copying and assigning to structures

```
struct address a, b;
a.zip = 123456;
a.street = "street";
a.city = "recife";
b = a;
b.zip = 654321;
printf("%d, %s, %s\n", a.zip, a.street,
  a.city);
```

- prints

```
123456, street, recife
```

Structures and functions

- Structures can be passed as parameters to a function
- Structures are passed by value i.e. copying their content
- Large structures should be passed by reference by passing their pointers as parameters to functions

Pointers to structures

```
struct address * b;
b = (struct address *) malloc(sizeof(struct
  address));
b->street = "street";
b->city = "recife";
(*b).zip = 654321;
printf("%d, %s, %s\n", (*b).zip,
b->street, b->city);
```

- The `.` operator has higher precedence than the `*` operator
- C provides the operator `->` to facilitate the syntax for accessing members of structures through their pointers
- A structure can point to itself (e.g. in a tree structure)

Arrays of structures

- Declaration

```
struct address a[10];
```

- Initializers

```
struct address a[] = {"street1",
  "recife", 4123456, "street2",
  "salvador", 654321};
struct address a[] = {{"street1",
  "recife"}, {"street2", "salvador",
  654321}};
```

Typedef

- Used for creating new data types

```
typedef unsigned short UCHAR;
```

- New types using structures

```
typedef struct address {
   char * street;
   char * city;
   int zip;
} Address;
b = (Address *) malloc(sizeof(Address));
b->street = "street";
b->city = "recife";
b->zip = 654321;
```

Unions

- Looks like a structure but stores only one type at any given time
- The compiler assigns a union a size large enough to store the widest type

```
union number {
  int ival;
  float fval;
} n;
```

- Unions can be nested within structures
- Unions support the same operations as structures

Bit fields

- Useful for conveniently handling several option flags as a single entity
- Each flag field can only be be an `int`
- The fields cannot be arrays nor be pointed to (or thus have the & operator applied to them)

```
struct bit_fields {
  unsigned int is_keyword : 1;
  unsigned int is_extern : 1;
  unsigned int is_static : 1;
} f;
```

Exercise

- Write a program to count the occurrence of each word in a given string. Use a binary search tree to store the words along with their counts.
- Print the words with their count to standard output in an ascending order by traversing the binary search tree in-order.