York University
Electrical Engineering and Computer Science

EECS2031: Software Tools
SU2016
Assignment #9

Chapter 16: Exercises

3. (a) Show how to declare a tag named `complex` for a structure with two members, `real` and `imaginary`, of type `double`.

   (b) Use the `complex` tag to declare variables named `c1`, `c2`, and `c3`.

   (c) Write a function named `make_complex` that stores its two arguments (both of type `double`) in a `complex` structure, then returns the structure.

   (d) Write a function named `add_complex` that adds the corresponding members of its arguments (both `complex` structures), then returns the result (another `complex` structure).

(a)

```
struct complex {
  double real, imaginary;
};
```

(b) `struct complex c1, c2, c3;`

(c)

```
struct complex make_complex(double real, double imaginary)
{
  struct complex c;

  c.real = real;
  c.imaginary = imaginary;
  return c;
}
```

(d)

```
struct complex add_complex(struct complex c1, struct complex c2)
{
  struct complex c3;

  c3.real = c1.real + c2.real;
  c3.imaginary = c1.imaginary + c2.imaginary;
  return c3;
}
```

5. Write the following functions, assuming that the `date` structure contains three members: `month`, `day`, and `year` (all of type `int`).

(a) `int day_of_year(struct date d);`

Returns the day of the year (an integer between 1 and 366) that corresponds to the date `d`.

(b) `int compare_dates(struct date d1, struct date d2);`

Returns –1 if `d1` is an earlier date than `d2`, +1 if `d1` is a later date than `d2`, and 0 if `d1` and `d2` are the same.

(a)

```
int day_of_year(struct date d)
{
  int day, month, days[] = {0, 31, 28, 31, 30, 31, 30, 31, 31, 30, 31, 30,
31};

  /* leap year adjustment */
  if ((d.year % 4 == 0) && (d.year % 100 != 0 || d.year % 400 == 0))
    days[2]++;

  day = d.day;
  for (month = 1; month < d.month; month++)
    day += days[month];
  return day;
}
```

(b)

```
int compare_dates(struct date d1, struct date d2)
{
  if (d1.year != d2.year)
    return d1.year < d2.year ? -1 : 1;
  if (d1.month != d2.month)
    return d1.month < d2.month ? -1 : 1;
  if (d1.day != d2.day)
    return d1.day < d2.day ? -1 : 1;

  return 0;
}
```

10. The following structures are designed to store information about objects on a graphics screen:

```
struct point { int x, y; };
struct rectangle { struct point upper_left, lower_right; };
```

A `point` structure stores the *x* and *y* coordinates of a point on the screen. A `rectangle` structure stores the coordinates of the upper left and lower right corners of a rectangle. Write functions that perform the following operations on a `rectangle` structure `r` passed as an argument:

(a) Compute the area of `r`.
(b) Compute the center of `r`, returning it as a `point` value. If either the *x* or *y* coordinate of the center isn't an integer, store its truncated value in the `point` structure.
(c) Move `r` by `x` units in the *x* direction and `y` units in the *y* direction, returning the modified version of `r`. (`x` and `y` are additional arguments to the function.)
(d) Determine whether a point `p` lies within `r`, returning `true` or `false`. (`p` is an additional argument of type `struct point`.)

(a)

```
int area(struct rectangle r)
{
  return (r.lower_right.x - r.upper_left.x) *
         (r.lower_right.y - r.upper_left.y);
}
```

(b)

```
struct point center(struct rectangle r)
{
  struct point c;

  c.x = (r.upper_left.x + r.lower_right.x) / 2;
  c.y = (r.upper_left.y + r.lower_right.y) / 2;
  return c;
}
```

(c)

```
struct rectangle move(struct rectangle r, int x, int y)
{
  struct rectangle r1 = r;

  r1.upper_left.x += x;
  r1.upper_left.y += y;
  r1.lower_right.x += x;
  r1.lower_right.y += y;
  return r1;
}
```

(d)

```
bool inside(struct rectangle r, struct point p)
{
  return r.upper_left.x <= p.x && p.x <= r.lower_right.x &&
         r.upper_left.y <= p.y && p.y <= r.lower_right.y;
}
```

2. Modify the `inventory.c` program of Section 16.3 so that the `p` (print) operation displays the parts sorted by part number.

```
void print(void)
{
  int i, pos, prev_part_number = 0, num_printed;

  printf("Part Number   Part Name                  "
         "Quantity on Hand\n");

  for (num_printed = 0; num_printed < num_parts; num_printed++) {

    /* find any part that hasn't already been printed */
    for (i = 0; i < num_parts; i++)
      if (inventory[i].number > prev_part_number) {
        pos = i;
        break;
      }
    /* find the part with the smallest number that hasn't
       already been printed */
    for (; i < num_parts; i++)
      if (inventory[i].number < inventory[pos].number &&
          inventory[i].number > prev_part_number)
        pos = i;

    printf("%7d       %-25s%11d\n", inventory[pos].number,
           inventory[pos].name, inventory[pos].on_hand);

    prev_part_number = inventory[pos].number;
  }
}
```

Chapter 17: Exercises

1. Having to check the return value of `malloc` (or any other memory allocation function) each time we call it can be an annoyance. Write a function named `my_malloc` that serves as a "wrapper" for `malloc`. When we call `my_malloc` and ask it to allocate `n` bytes, it in turn calls `malloc`, tests to make sure that `malloc` doesn't return a null pointer, and then returns the pointer from `malloc`. Have `my_malloc` print an error message and terminate the program if `malloc` returns a null pointer.

```
void *my_malloc(size_t n)
{
  void *p;

  p = malloc(n);
  if (p == NULL) {
    printf("Memory allocation failed\n");
    exit(EXIT_FAILURE);
  }
  return p;
}
```

3. Write the following function:

   ```
   int *create_array(int n, int initial_value);
   ```

   The function should return a pointer to a dynamically allocated `int` array with `n` members, each of which is initialized to `initial_value`. The return value should be `NULL` if the array can't be allocated.

```
int *create_array(int n, int initial_value)
{
  int *a, *p;

  a = malloc(n * sizeof(int));
  if (a != NULL)
    for (p = a; p < a + n; p++)
      *p = initial_value;
  return a;
}
```

Chapter 17: Programming Projects

1. Modify the `inventory.c` program of Section 16.3 so that the `inventory` array is allocated dynamically and later reallocated when it fills up. Use `malloc` initially to allocate enough space for an array of 10 `part` structures. When the array has no more room for new parts, use `realloc` to double its size. Repeat the doubling step each time the array becomes full.