Nach dem OpenEyes sehr schön erklärt hat, wie das Programm insgesamt arbeitet, versuche ich hier einmal zu erklären, wie das funktioniert. Um die Funktionsweise zu verstehen, lassen wir die Rekursion einfach mal weg, d.h. wir betrachten einfach mal nur eine Instanz.

Als Quelle habe ich den von SourceCode von http://www.linux-related.de/index.html?/coding/sort/sort_merge.htm genommen, auch wenn der nicht standalone funktioniert.

Hier noch einmal der komplette Sourcecode:

```
void mergesort(int a[], int l, int r){        // l=linker Rand, r=rechter Rand
    if(r>l){
        int i, j, k, m;                       // Variablen deklarieren
        m=(r+l)/2;                            // Mitte ermitteln
        mergesort(a, l, m);                   // linke Teildatei
        mergesort(a, m+1, r);                 // rechte Teildatei
        for(i=m+1; i>l; i--) b[i-1]=a[i-1];   // linke  Teildatei in Hilfsarray
        for(j=m; j<r; j++) b[r+m-j]=a[j+1];   // rechte Teildatei umgedreht in
                                              // Hilfsarray
        for(k=l; k<=r; k++)
            a[k]=(b[i]<b[j])?b[i++]:b[j--];   // füge sortiert in a ein
    }                                         // Ende der if-Abfrage
}                                             // Ende der Funktion
```

Fangen wir mit der ersten Zeile an:

```
void mergesort(int a[], int l, int r)
```

`void mergesort()` bedeutet, dass jetzt eine Funktion `()` mit dem Namen `mergesort` kommt, die keinen Rückgabewert hat, deshalb `void`.

In Klammern wird angegeben, welche Parameter die Funktion erwartet. Das ist einmal ein Array, d.h. eine Liste [] gleicher Elemente, hier `int`, also Integer, die innerhalb der Funktion mit dem Namen a angesprochen wird. Außerdem noch zwei weitere Integer, die innerhalb der Funktion mit `l` und `r` angesprochen werden. Sie stehen, wie wir gleich noch sehen werden, für den linken und rechten Rand des Arrays, der sortiert werden soll.

Hierbei ist jetzt eine Besonderheit zu erwähnen (die zum vollen Verständnis unabdingbar ist): Variablen können entweder **by value** oder **by reference** übergeben werden. Was bedeutet das?

By value bedeutet, dass der Wert der Variablen übergeben wird. Wenn ich die Funktion mit mergesort(Array[], links, rechts) aufrufe, wird zur Laufzeit (d.h. wenn das Programm ausgeführt wird) geschaut, welcher Wert sich in den Variablen links und rechts befindet. Nehmen wir für unser Beispiel mal die Werte 0 für links und 7 für rechts an, hätten wir also auch schreiben können mergesort(Array[], 0, 7) Die Funktion mergesort speichert die übergebenen Werte in den lokalen (d.h. nur für die Funktion selbst, genau genommen sogar nur für diese Instanz der Funktion, sichtbare) Variablen l und r.[1]

By reference bedeutet, das nicht der Inhalt/Wert der Variable, sondern lediglich die (physikalische)[2] Speicheradresse der Variable weitergegeben wird. Deshalb ist es für die Funktion wichtig zu wissen, welche Art von Array das ist, weil davon die sogenannte Pointerarithmetik (also die Berechnung bestimmter Speicherstellen) abhängig ist. Das Entscheidende dabei aber ist, das by reference übergebene Variablen innerhalb der Funktion geändert werden können und diese Änderungen dann auch außerhalb der ändernden Funktion sichtbar sind, da der physikalische Speicher verändert wurde. Verschiebt also irgendeine Instanz von mergesort den Inhalt der Variablen a[1] nach z.B. a[0] (wie das mit dem a von caroline passiert), dann ist diese Änderung in jeder Instanz sichtbar, da alle auf den gleichen Speicherbereich zugreifen. Im Gegensatz dazu würden Änderungen an l und r nichts an den aufrufenden links und rechts ändern.

---

[1] Ruft mergesort im Verlauf wieder mergsort auf, heißt das, dass eine neue Instanz geschaffen wird. Auch sie hat wieder lokale Variablen l und r, die zwar genauso heißen, aber nur innerhalb dieser Instanz bekannt sind. Sprich sie verweisen auf einen physikalisch anderen Speicherplatz als l und r der aufrufenden Instanz. Jede Instanz kennt immer nur SEIN l und r, deshalb lokale Variable, d-h. nur innerhalb der Funktion sichtbar.

[2] In Klammern, weil sich streng genommen hinter den Adressen verschiedener Variablen auch der gleiche Speicherbereich verbergen kann, aber das lassen wir mal außen vor…

Wir wissen jetzt also, es gibt eine Funktion mergesort die auf einen gemeinsamen Speicherbereich zugreift und zwei lokale Variablen als Grenzwerte erhält.

Die geschweifte Klammer `{` am Ende der Zeile bedeutet lediglich, dass jetzt die eigentlichen Programmangaben für die Funktion anfangen. Je nach Programmierer wirst du die Klammer auch in der Zeile darunter finden. Das ist letztlich eine Philosophiefrage. Inhaltlich ist es egal, solange der sogenannte Anweisungsblock von geschweiften Klammern eröffnet und geschlossen wird.

Die nächste Zeile

```
if(r>l){
```

fragt einfach nur ab, ob nicht der linke und rechte Rand gleich sind, sprich ob es mehr als 1 Element gibt. (Siehe Openeyes und meine Erklärung im Thread, 1 Element ist immer sortiert, d.h. es muss nichts weiter getan werden.) Auch hier findet sich wieder die Eröffnung eines so genannten Anweisungsblocks. Sprich alles innerhalb der geschweiften Klammern[3] wird nur ausgeführt, wenn die Bedingung rechter Rand größer linker Rand (`r>l`) zutrifft.

Weiter im Sourcecode:

```
int i, j, k, m;
```

Hier werden, wie die sogenannten Inlinekommentare (die immer mit // beginnen) erläutern ein paar Variablen definiert (sprich Speicherplatz reserviert und Namen zugewiesen). Es sind allesamt Integer, sprich sie können alle einen Wert zwischen 0 und 255 annehmen.[4] Wozu sie dienen werden wir gleich noch sehen.

Mit der Zeile

```
m=(r+l)/2;
```

Wird das Element in der Mitte ermittelt. Und das einfach, in dem der rechte mit dem linken „Rand“ addiert wird und dann durch 2 geteilt wird. Man stelle sich einfach die folgenden Felder vor:

0	1	2	3	4	5	6	7
c	a	r	o	l	i	n	e

Das Array A

Die Zahlen stellen die Adressen der Speicherstellen dar. Wir haben also 8 Elemente mit den Adressen 0 bis 7. Wenn `mergesort` zum ersten Mal aufgerufen wird, dann ist `l`=0 und `r`=7. `r+l` ist 7, 7/2 ist… 3 (!)

Da mit Integern gerechnet wird, die nur ganze Zahlen aufnehmen können, wird ein evtl. Nachkommawert gnadenlos abgeschnitten. Damit ist `m` also 3.

In den nächsten beiden Zeilen

```
mergesort(a, l, m);           // linke Teildatei
mergesort(a, m+1, r);         // rechte Teildatei
```

macht `mergesort` nichts anderes als dafür zu sorgen, dass alle Element von Links `l` = 0 bis zu Mitte `m`=3 und ab dem Element rechts neben der Mitte `m+1`=4 bis Rechts `r`=7 getrennt sortiert werden.

Wie das im Einzelnen passiert ist, denke ich grundsätzlich klar, weshalb wir hier jetzt einfach davon ausgehen, dass wir die Bereiche also in sich sortiert zurückerhalten.

---

[3] Klammern werden dabei durchaus ineinander verschachtelt. Einer der Gründe, warum ich die öffnenden Klammer lieber in eine einzelnen Zeile schreibe, ich habe sie dann auf einer Ebene mit den schließenden…

[4] Streng genommen kann man das so genau gar nicht sagen. Ob ein Int wirklich nur 1 Byte ist oder vielleicht 2 oder gar 4 ist abhängig vom Rechnersystem. Ich gehe der Einfachheit halber aber mal von 1 Byte aus.

(Wir wissen aber jetzt, dass der erste aufgerufene mergesort mit den Werten 0 und 3 rechnet und selber wieder mergesort mit m=1 (0 + 3) / 2 = 1 (Integer!) aufruft. Also einmal die Elemente 0 – 1 und einmal 2 -3 sortieren lässt.)

Und der zweite mergesort? Wird mit 4 und 7 aufgerufen. Welchen Wert hat dort m? [5])

Unser Array sieht jetzt also folgendermaßen aus (siehe auch die Erläuterungen von OpenEyes):

0	1	2	3	4	5	6	7
a	c	o	r	e	i	l	n

Das Array A in der ersten Instanz, beide Teilarray bereits sortiert

Zur Erinnerung, in dieser Instanz von mergesort rechnen wir mit l=0, r=7 und m=3

Jetzt kommt etwas tricky stuff. Bzw. wir müssen uns eben über For-Schleifen unterhalten. For-Schleifen sind sogenannte Zählschleifen. D.h. sie zeichnen sich dadurch aus, dass ich einen Zähler habe, der einen bestimmten Startwert erhält, mit diesem irgendwas veranstalte (was, schauen wir uns gleich an) und ihn danach verändere (nach oben oder nach unten). Und das solange, wie ich eine bestimmte Bedingung erfülle. D.h. die Zeile

```
for(i=m+1; i>l; i--) b[i-1]=a[i-1]; // linke  Teildatei in Hilfsarray
```

besagt, das `i` (der Zähler bei `m+1` (also… 4) beginnen soll. Laufen soll die Schleife solange `i` größer als der linke Rand `l` ist (also >0). Nach der Verarbeitung soll i um 1 vermindert werden. (`i--` => `i=i-1`) Und was soll gemacht werden? Das steht (da es sich dabei nur um eine einzige Anweisung handelt) direkt hinter dem so genannten Schleifenkopf.[6]

Dem Element `i - 1` von `b[ ]` soll der Wert des Elementes `i - 1` von `a[ ]` zugewiesen werden. Sprich, es wird einfach eine Kopie des linken Arraybereiches gemacht. [7]

Begonnen wird damit in der Mitte. `i` hat am Anfang den Wert 4, zugegriffen wird aber auf `i - 1`, d.h. auf 3. Warum ein solches? Die Abfrage der Bedingung lautet: solange `i` größer `l` ist. D.h. die Schleife wird das letzte mal mit i=1 ausgeführt, dabei wird dann auf das Element 0 zugegriffen (weil eben wieder i-1). Es werden also genau die Elemente a[3], a[2], a[1] und a[0] nach b[3], b[2],b[1] und b[0] kopiert. Damit ergibt sich folgendes Gesamtbild:

0	1	2	3	4	5	6	7
a	c	o	r	e	i	l	n

Das Array A in der ersten Instanz, beide Teilarray bereits sortiert

0	1	2	3	4	5	6	7
a	c	o	r				

Das Array B in der ersten Instanz, „linkes“ Teilarray kopiert

Das war noch verhältnismäßig einfach, da die Werte einfach 1:1 an die gleichen Positionen kopiert wurden. Wichtig noch: i ist nach dem letzten Kopiervorgang erneut dekrementiert worden, ist also jetzt 0. Damit war die Bedingung i>l nicht mehr erfüllt, da ja beide Werte 0 sind. Wir merken uns also: i ist 0 und zeigt damit quasi auf das ganz linke Element.

Jetzt wird es lustig:

```
for(j=m; j<r; j++) b[r+m-j]=a[j+1];    // rechte Teildatei umgedreht in
```

[5] 7 + 4 = 11, 11 / 2 = 5 (!), ruft also wieder mergesort mit 4 und 5 bzw. 6 und 7 auf.

[6] Hier könnte auch wieder ein Anweisungsblock in { } stehen. Spätestens wenn der über mehrere Zeilen geht, weiß man, warum das Ding Schleifenkopf heißt.

[7] Solche Bezeichnungen wie links und rechts im Zusammenhang mit Arrays darf man bitte nicht zu wörtlich nehmen. Das ist so ein wenig wie mit den Karten und den Himmelsrichtungen. Auf der Karte ist der Westen links von der Mitte. Deshalb ist aber der Süden nicht unten und der Norden nicht oben. Es geht hier nur um die bessere Orientierung in der zeichnerischen Darstellung.

```
// Hilfsarray
```

Auch hier wieder eine Zählschleife. Ihr Zähler ist j und beginnt mit dem Startwert m, also 3. Die Schleife soll solange laufen, wie der Zähler kleiner r ist und nach jedem Durchlauf um 1 erhöht werden. Sprich die Schleife läuft von der Mitte bis an den rechten Rand. Von der Mitte? Aber das Element [3] gehört doch in den linken Bereich... Gemach!

Die Anweisung der Zählschleife ist wieder eine einfache Zuordnung. Die Werte aus a[3+1] also a[4] (jetzt sind wir wieder in der richtigen Hälfte!) werden nacheinander in ein Feld von b[ ] kopiert. (Und wir sehen auf Anhieb, j soll laufen solange es kleiner r ist, der letzte Wert von j ist also 6, womit klar ist, dass zuletzt das Element a[6+1] also a[7] kopiert wird. Genauso haben wir das erwartet...)

Aber in welches Element von b[ ] werden die Werte jeweils kopiert? Rechnen wir es einfach aus: `r + m – j`, (zur Erinnerung: r = 7, m = 3 und j haben wir gerade auf m, also 3 gesetzt). Der erste Wert wird also in b[7+3 -3] also b[7] kopiert. Hm... rechnen wir noch einmal mit dem letzten Wert der Zählschleife: da wird der Wert aus a[6+1] in das Feld b[7 + 3 – 6] also b[4] kopiert. Aha! Durch die Berechnung wird also die Reihenfolge von links nach rechts vertauscht...

Die beiden Arrays sehen jetzt also so aus:

0	1	2	3	4	5	6	7
a	c	o	r	e	i	l	n

Das Array A in der ersten Instanz, beide Teilarray bereits sortiert

0	1	2	3	4	5	6	7
a	c	o	r	n	l	i	e

Das Array B in der ersten Instanz, „linkes“ und „rechtes“ Teilarray kopiert

Beim letzten Durchgang wurde j noch einmal um eins erhöht, hat jetzt also den Wert 7. Merken wir uns wieder.

Und nun kommt natürlich die Frage aller Fragen: Wat soll dä driess? Hochdeutsch: Und wozu das ganze? Sehen wir gleich.

Sehen wir uns die letzte For-Schleife an:

```
for(k=l; k<=r; k++)
```

Wieder ein Zähler, diesmal k, der mit l, also 0 initiiert wird uns solange laufen soll, wie er kleiner oder gleich r, also 7 ist.

Nach jedem Durchgang wird k um 1 erhöht. Wir sehen also sofort, dass k einmal durch das komplette Array von 0 bis 7 läuft. Und was macht die Schleife? Das sehen wir in der Anweisung:

```
a[k]=(b[i]<b[j])?b[i++]:b[j--];     // füge sortiert in a ein
```

Wir sehen links a[k]= , d.h. hier werden also offensichtlich den Elementen des Array nacheinander „irgendwelche“ Werte zugewiesen. Und welche? Jetzt wird es wieder tricky. Hier gibt es zwei Dinge zu wissen:

1. `(b[i]<b[j])?b[i++]:b[j--];` ist eine sogenanntes Inline-If, das einfach eine Bedingung abprüft, hier ob b an der Stelle i (b[i]) kleiner ist als b an der Stelle j (b[j]) Ist das der Fall, wird der Wert b[i] zurückgegeben. Ist das nicht der Fall, wird der Wert an der Stelle b[j] zurückgegeben. i und j? Aber da stehen doch i++ und j++? Richtig. Damit kommen wir zu
2. Steht irgendwo eine Variable gefolgt von einem ++ (oder --)wird die Variable zunächst unverändert genommen und erst danach in- (oder de-)krementiert.

OK, wir hatten uns gemerkt, dass i auf 0 und j auf 7 stand. Und jetzt verstehen wir auch warum die Werte umgedreht wurden. (Ein Nachtrag kommt gleich noch.) Wir gehen von außen nach innen. Die jeweils kleinsten Werte stehen außen, die jeweils größten innen.

Und jetzt langsam:

1. Schleifendurchlauf: k ist 0, i ist 0 und j ist 7
   Wir vergleichen b[0]='a' mit b[7]='e'. 'a' ist kleiner 'e', also wird a[0] das 'a' aus b[0] (i) zugewiesen. Anschließend wird i um eins erhöht. i zeigt jetzt nicht mehr auf das 'a' in b[0] sondern auf das 'c' in b[1].

2. Schleifendurchlauf: k ist 1, i ist 1 und j ist 7
   Wir vergleichen b[1]='c' mit b[7]='e'. 'c' ist kleiner 'e', also wird a[1] das 'c' aus b[1] (i) zugewiesen. Anschließend wird i um eins erhöht. i zeigt jetzt nicht mehr auf das 'c' in b[1] sondern auf das 'o' in b[2].

3. Schleifendurchlauf: k ist 2, i ist 2 und j ist 7
   Wir vergleichen b[2]='o' mit b[7]='e'. 'o' ist nicht kleiner 'e', also wird a[2] das 'e' aus b[7] (j) zugewiesen. Anschließend wird j um eins erniedrigt. j zeigt jetzt nicht mehr auf das 'e' in b[7] sondern auf das 'i' in b[6].

4. Schleifendurchlauf: k ist 3, i ist 2 und j ist 6
   Wir vergleichen b[2]='o' mit b[6]='i'. 'o' ist nicht kleiner 'i', also wird a[3] das 'i' aus b[6] (j) zugewiesen. Anschließend wird j um eins erniedrigt. j zeigt jetzt nicht mehr auf das 'i' in b[6] sondern auf das 'l' in b[5].

5. Schleifendurchlauf: k ist 4, i ist 2 und j ist 5
   Wir vergleichen b[2]='o' mit b[5]='l'. 'o' ist nicht kleiner 'l', also wird a[4] das 'l' aus b[5] (j) zugewiesen. Anschließend wird j um eins erniedrigt. j zeigt jetzt nicht mehr auf das 'l' in b[5] sondern auf das 'n' in b[4].

Wir sehen also schon ganz gut, wie wir uns von außen nach innen hangeln und dabei immer den nächstkleineren Wert zurück nach a[ ] schreiben. Schauen wir uns den nächsten Durchlauf auch noch an.

6. Schleifendurchlauf: k ist 5, i ist 2 und j ist 4
   Wir vergleichen b[2]='o' mit b[4]='n'. 'o' ist nicht kleiner 'n', also wird a[5] das 'n' aus b[4] (j) zugewiesen. Anschließend wird j um eins erniedrigt. j zeigt jetzt nicht mehr auf das 'n' in b[4] sondern auf das 'r' in b[3].

Jetzt verstehen wir auch vollständig, was das umdrehen der Reihenfolge in dem Hilfsarray sollte. Es wäre natürlich möglich gewesen, das ganze so zu schreiben, das i ganz links anfängt und j in der Mitte. Aber was wenn alle Werte (so wie hier auf einer Seite abgearbeitet sind? Durch die Umkehrung verweist j jetzt auf das größte Element von links. Raffiniert, oder?)

In den letzten beiden Durchläufen werden jetzt das 'o' mit dem 'r' verglichen. Mit dem erwarteten Ergebnis, dass erst das 'o' und dann das 'r' in a[6] und a[7] einsortiert werden. (Wenn man den Weg weiterverfolgt, stellt man fest, dass im letzten Durchlauf beide Indizes (so nennt man die Variablen die die Positionen in einem Array bestimmen) den Wert 3 haben. OK, vergleichen wir halt 'r' mit 'r'. Das Programm entscheidet sich anschließend spontan dazu … richtig, 'r' nach a[7] zu kopieren.

Fertig!

So sehen unsere Array jetzt aus:

0	1	2	3	4	5	6	7
a	c	e	i	l	n	o	r

Das Array A in der ersten Instanz, beide Teilarray gemerged, sprich gemischt

0	1	2	3	4	5	6	7
a	c	o	r	n	l	i	e

Das Array B in der ersten Instanz, „linkes“ und „rechtes“ Teilarray kopiert

Die beiden } in den letzten beiden Zeilen machen nichts anderes als einmal den if-Anweisungsblock zu schließen, d.h. der gesamte Block wird nur ausgeführt, wenn die Bedingung erfüllt ist, und die Definition der Funktion mergesort zu beenden.

Was hier für die Hauptinstanz von mergesort beschrieben ist, funktioniert genau so für jede Unterinstanz. Einfach mal durchtesten...

Ich erwähnte eingangs, dass der Sourcecode so alleine nicht lauffähig ist. Das hängt eben damit zusammen, das wir hier zwar von einem Aufruf der Funktion mit einem Array und den weiteren Parametern 0 und 7 sowie der Existenz eines Hilfsarrays b[ ] ausgegangen sind, dies aber tatsächlich nicht vorhanden ist.