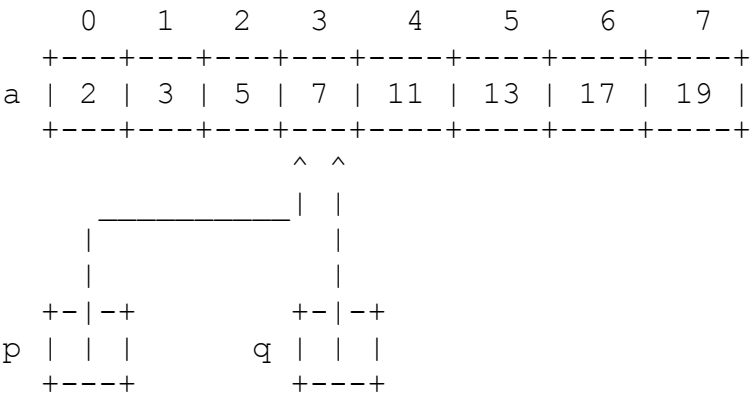
Wenn man das erste Mal mit Zeigern in Kontakt kommt, sieht man oft den Wald vor lauter Bäumen nicht mehr. Solange man noch kein intuitives Gefühl für Zeiger entwickelt hat, empfiehlt es sich dringend, für alle an einem Programm beteiligten Variablen kleine Boxen zu zeichnen. Dann kann man Zeiger recht gut mit Pfeilen veranschaulichen. Beispiel:

```
int a[8] = {2, 3, 5, 7, 11, 13, 17, 19};
int * p = &a[3];
int * q = p;
```



Aufgabe 4.1 Zahlen sortieren

Schreibe eine Funktion `void sort2(int * a, int * b)`, welche die beiden Zahlen *aufsteigend* sortiert. Sprich, wenn man `sort2(&x, &y)` aufruft, soll anschließend $x \leq y$ gelten, egal wierum `x` und `y` vorher sortiert waren. Schreibe einen Test, der `sort2` einmal mit aufsteigend sortierten Zahlen und einmal mit absteigend sortierten Zahlen aufruft.

Schreibe eine weitere Funktion `sort3` zum Sortieren von drei Zahlen und teste diese ausführlich. Hier gibt es deutlich mehr mögliche Eingaben, z.B. 13, 11, 17 und 17, 11, 13.

Aufgabe 4.2 Zeiger und Zeichenketten

Analysiere die folgende Funktion. Was bedeuten die beiden Parameter? Was liefert sie zurück? Was bedeutet `*s` als erste Teilbedingung? Welche Auswirkung hat `++s`?

```
char * my_strchr(char * s, char c)
{
    while (*s && *s != c) ++s;
    return s;
}
```

Analysiere den folgenden Quelltext. Warum kann man das Array `a` (vom Typ `char[13]`) an `my_strchr` übergeben, obwohl ein Zeiger (vom Typ `char *`) erwartet wird?

```
char a[] = "Hallo Welt.\n";
char * p = my_strchr(a, 'l');
printf("%p - %p = %d (%c)\n", p, a, p - a, *p);
```

Was wird an `printf` übergeben? Was bedeutet `*p`? Was bedeutet `p-a`? Warum kann man den Formatstring `%p` offenbar nicht nur für Zeiger, sondern auch für Arrays verwenden?

Aufgabe 4.3 Effizientes Rotieren mit drei Umdrehungen

Schreibe eine Funktion `void rotate_left(char * s, int n)`, welche die Zeichenkette `s` um `n` Zeichen nach links rotiert. Die Zeichenkette `"wurstbrot"` soll für n=5 beispielsweise auf die Zeichenkette `"brotwurst"` abgebildet werden.

Mit einem zweiten lokalen Array ließe sich die Aufgabe mit linearem Speicheraufwand lösen. Es geht aber auch mit konstantem Speicheraufwand (und linearem Zeitaufwand)!

Ansatz: rotate(α β) = reverse(reverse(α) reverse(β)) = β α für Teilstrings α und β

Beispiel: rotate(abc xy) = reverse(reverse(abc) reverse(xy)) = reverse(cba yx) = xy abc

Die einzelnen Schritte des Algorithmus sind im Folgenden noch einmal veranschaulicht:

```
wurstbrot   /* Eingabe */
tsruwbrot   /* Linkes Umdrehen */
tsruwtorb   /* Rechtes Umdrehen */
brotwurst   /* Gesamtes Umdrehen */
```

Aufgabe 4.4 Zeichenketten vertauschen

Mache Dir auf Papier klar, was das folgende Programm tut:

```
void swap_strings(char * p, char * q)
{
    char * t = p;
    p = q;
    q = t;
}

int main()
{
    char * a = "world";
    char * b = "hello";
    swap_strings(a, b);
    printf("%s %s!\n", a, b);
    return 0;
}
```

Probiere das Programm anschließend aus. Entspricht das Ergebnis Deinen Erwartungen? Kannst Du das Programm so korrigieren, dass es Deinen Erwartungen entspricht?
Falls Du damit Schwierigkeiten hast, schreibe als erste Zeile in das Programm:

```
typedef char * string;
```

und ersetze in den restlichen Zeilen darunter alle Vorkommen von `char*` durch `string`, dann sollte das Problem (und hoffentlich auch dessen Lösung) offensichtlich sein.

Das Programm funktioniert nicht mehr, wenn Du die Zeiger `a` und `b` durch Arrays ersetzt. (Warum?) Wieder ein schönes Beispiel dafür, dass Arrays und Zeiger nicht dasselbe sind...

Aufgabe 4.5 Kommandozeilen-Argumente

Eine zweite legale Signatur für `main` lautet: `int main(int argc, char * * argv)`. Der erste Parameter gibt dabei die Anzahl der Argumente an, und der zweite Parameter zeigt auf das erste Element eines Arrays von Zeigern. Jeder dieser Zeiger zeigt wiederum auf das erste Zeichen einer Zeichenkette. Veranschauliche Dir diese Struktur auf Papier!

Schreibe dann ein Programm, das seine Argumente zeilenweise auf der Konsole ausgibt. Diese Argumente kann man wie folgt in Visual Studio festlegen:

Projekt > Projekt-Eigenschaften...

Konfigurationseigenschaften > Debugging

Befehlsargumente: Fred loves pineapple juice

Aufgabe 4.6 Bibliotheksfunktionen nachbauen

Implementiere die folgenden Funktionen, **nach Möglichkeit nur mit Zeiger-Operationen wie *s und s++ anstatt mit Index-Operationen wie s[i] und i++**, und teste sie ausführlich:

```
/* Bestimmt die Länge der Zeichenkette s. */
int my_strlen(const char * s);

/* Überschreibt dst mit src. */
void my_strcpy(char * dst, const char * src);

/* Hängt src hinten an dst ran. */
void my_strcat(char * dst, const char * src);

/* Liefert irgendeine negative Zahl, falls s in einem Wörterbuch
vor t erscheinen würde, irgendeine positive, falls s nach t käme,
bzw. die Zahl 0, wenn s und t dasselbe Wort repräsentieren. */
int my_strcmp(const char * s, const char * t);

/* Liefert einen Zeiger auf das erste Vorkommen von t in s. */
char * my_strstr(char * s, const char * t);
```

Der Aufrufer von `my_strcpy` und `my_strcat` muss selber daran denken, genügend Speicherplatz in `dst` zur Verfügung zu stellen. Ansonsten entsteht ein sog. *Pufferüberlauf*. Pufferüberläufe gehören zu den kritischsten Sicherheitsproblemen in C.

Schreibe zwei weitere Funktionen `my_strncpy` und `my_strncat`, die als zusätzlichen Parameter die Größe des Ziel-Arrays akzeptieren und daher Pufferüberläufe verhindern können, indem der Kopiervorgang einfach abgebrochen wird, sobald das Ende erreicht ist.

```
/* Überschreibt dst mit src, garantiert ohne Pufferüberlauf. */
void my_strncpy(char * dst, const char * src, int dst_capacity);

/* Hängt src hinten an dst ran, garantiert ohne Pufferüberlauf. */
void my_strncat(char * dst, const char * src, int dst_capacity);
```