3. Übungsblatt:
Programmierpraktikum II (SS 2002)

Abgabe: 3. Juni 2002

Erlaubte Bibliotheken:

- `stdio.h`

Makefile

Zur Abgabe gehört ab diesem Blatt auch ein Makefile. Für jede bearbeitete Programmieraufgabe muss das Makefile einen Eintrag enthalten, der das Compilieren des Programmcodes mit einem einfachen `make`-Befehl ermöglicht.

Bsp.: Wenn du Aufgabe 1b) bearbeitet hast, dann muss dein Makefile einen Eintrag `aufgabe1b` enthalten. Dieser muss so gestaltet sein, dass der Linux-Befehl `make aufgabe1b` ein ausführbares Programm `aufgabe1b` erzeugt.

Hinweis: Beim Compilieren muss das `-Wall`-Flag unbedingt gesetzt sein!

Aufgabe 1: Berechnung des Binomialkoeffizienten (naiv)

(6 Punkte)

Gegeben sei das folgende Hauptprogramm, das für zwei positive ganze Zahlen n und k den Binomialkoeffizienten $\binom{n}{k} = \frac{n!}{k!(n-k)!}$ berechnet:

```
#include <stdio.h>
#include "fakultaet.h"

int main() {
  unsigned int n, k, s;
  printf("Gib n ein! "); scanf("%u", &n);
  printf("Gib k ein! "); scanf("%u", &k);
  if(k>n) {
    printf("Eingabefehler (k > n)!\n");
    return -1;  }
  s = fakultaet(n) / (fakultaet(k) * fakultaet(n-k));
  printf("n ueber k ist %i.\n", s);
  return 0;
}
```

a) Schreibe zunächst eine **nicht-rekursive** Funktion mit dem Prototypen
`unsigned int fakultaet(unsigned int),`
die bei Eingabe von n den Wert $n!$ zurückgibt. Welches ist der größte Wert von n, für den die Funktion das korrekte Ergebnis zurückliefert? Woran liegt das?

b) Schreibe das obige Hauptprogramm in eine Datei und lege für die Funktion `fakultaet()` geeignete Dateien `fakultaet.c` und `fakultaet.h` an. Füge einen Eintrag `aufgabe1b` in das Makefile ein.

c) Da in der obigen Implementierung nur sehr kleine Binomialkoeffizienten korrekt berechnet werden können, wollen wir nun statt ganzer Zahlen Fließkommazahlen verwenden, mit denen man ja bekanntlich viel größere Werte darstellen kann.
Kopiere die obigen Dateien unter anderem Namen und modifiziere sie so, dass als Datentyp jetzt `double` statt `unsigned int` verwendet wird. Füge einen Eintrag `aufgabe1c` in das Makefile ein.
Finde nun ein Beispiel für n und k (beides positive ganze Zahlen), für das auch dieses Programm falsche Werte liefert. Woran liegt das?

Aufgabe 2: Berechnung des Binomialkoeffizienten (clever)

(4 Punkte)

Da der naive Ansatz offenbar nicht funktioniert, soll nun ein besseres Verfahren verwendet werden. Wir berechnen dazu den Binomialkoeffizenten rekursiv nach der Regel

$$\binom{n}{k} = \frac{n-k+1}{k} \cdot \binom{n}{k-1} \quad ,$$

wobei $\binom{n}{0} = 1$ ist[1]. Schreibe ein Programm, das den Binomialkoeffizienten unter Ausnutzung dieser Äquivalenz **rekursiv** berechnet! Vergiss nicht, einen Eintrag `aufgabe2` in das Makefile einzufügen.

[1] Unbepunktete Zusatzaufgabe: Beweise die Korrektheit dieser Äquivalenz!

Aufgabe 3: Hamming-Code (Hilfsfunktionen)

(8 Punkte)

In Aufgabe 4 wollen wir den 12-Bit-Hamming-Code aus der Vorlesung *Praktische Informatik II* implementieren. Dazu benötigen wir zunächst einige Hilfsfunktionen, die wir in einer kleinen Bibliothek unterbringen wollen.

- Eine Funktion mit dem Prototypen
  `void displaybits(int)`,
  die einen Integer-Wert als Input erhält und die letzten 12 Bit (*least significant bits, LSB)* seiner Binärdarstellung am Bildschirm ausgibt. So müsste beispielsweise `displaybits(25)` die Bitfolge `000000011001` auf den Bildschirm schreiben.

- Eine Funktion mit dem Prototypen
  `int innerproduct(int, int)`,
  die zwei Integer-Werte a und b als Input erhält und diese als 12-Bit-Werte auffasst. Sie berechnet für $a = (a_{11}, \ldots, a_0)$ und $b = (b_{11}, \ldots, b_0)$ das innere Produkt
  $$c = a_{11}b_{11} \oplus \ldots \oplus a_1b_1 \oplus a_0b_0 \quad .$$
  Das Output nimmt dabei die Werte 0 oder 1 an.

- Eine Funktion mit dem Prototypen
  `void flipbit(int*, int)`,
  die bei Eingabe von Bitwerten a und b das b-te Bit von a invertiert, falls $0 \leq b \leq 11$ gilt. Andernfalls bleibt a unverändert.
  Bem.: Beachte, dass a per *Call by Reference* übergeben wird!
  Bsp.: Der Aufruf der Funktion mit `flipbit(377,6)` muss den Wert 313 ausgeben, da
  $$\begin{array}{rcl} 000101111001 & = & (377)_{10} \\ \oplus\, 000001000000 & = & (2^6)_{10} \\ =\, 000100111001 & = & (313)_{10} \end{array}$$
  ist.

Zur Abgabe gehören die Dateien `divlib.c` und `divlib.h` sowie eine Datei mit dem Hauptprogramm. Dieses soll jede Funktion aus `divlib.c` für drei verschiedene Inputs aufrufen und zeigen, dass sie die obigen Anforderungen erfüllt. Vergiss auch nicht, einen entsprechenden Eintrag `aufgabe3` in das Makefile einzufügen!

Aufgabe 4: Hamming-Code (Kernfunktionen)

(12 Punkte)

Und jetzt kann's losgehen. Implementiert werden soll der (8,4)-Hammingcode aus der Vorlesung *Praktische Informatik II* (Folien 2-62 bis 2-64). Dazu benötigen wir die folgenden Kernfunktionen (die wiederum die obigen Hilfsfunktionen benutzen dürfen):

- `int encode(char)`:
  Diese Funktion erhält als Input ein 8-Bit-Datenwort $d = (d_7, \ldots, d_0)$ und liefert das dazugehörige Codewort

  $$w = (d_7, d_6, d_5, d_4, c_3, d_3, d_2, d_1, c_2, d_0, c_1, c_0)$$

  zurück wie auf Folie 2-63 beschrieben.
  Bem.: Beachte, dass c_0 hier als *least significant bit* gespeichert wird!

- `void correct(int*)`:
  Diese Funktion erhält als Input (*Call by Reference*) einen 12-Bit-Wert. Falls dieser einen Fehler enthält (d.h., falls es sich nicht um ein korrektes Codewort handelt), so soll der Fehler korrigiert werden wie auf Folie 2-64 beschrieben.

- `char decode(int)`:
  Diese Funktion erhält als Input ein korrektes 12-Bit-Codewort und liefert den zugehörigen 8-Bit-Datenwert zurück.

Diese Funktionen sind in den Dateien `hamming.c` und `hamming.h` zu beschreiben. Natürlich muss das Makefile einen Eintrag `aufgabe4` enthalten, der aus den Bibliotheksdateien `hamming.c`, `hamming.h`, `divlib.c`, `divlib.h` und dem Hauptprogramm ein lauffähiges Programm macht.
Das Hauptprogramm selbst soll wie folgt aussehen:

1. Der Nutzer gibt ein ASCII-Zeichen ein, das Programm ermittelt das zugehörige 12-Bit-Codewort und gibt dessen Binärdarstellung am Bildschirm aus.

2. Der Nutzer gibt einen ganzzahligen Rauschwert r ein (Werte zwischen $0 \leq r \leq 11$ bedeuten, dass das r-te Bit des Codeworts verrauscht wird, alle anderen Eingabewerte bedeuten, dass das Codewort unverändert bleibt). Das Programm gibt die Binärdarstellung des verrauschten Codeworts am Bildschirm aus.

3. Das Programm korrigiert das verrauschte Codewort und gibt die korrekte 12-Bit-Darstellung am Bildschirm aus.

4. Das Programm ermittelt den zugehörigen 8-Bit-Datenwert und gibt dessen ASCII-Darstellung am Bildschirm aus.