Stefan Kurtz **Universität Hamburg**
Abteilung für Genominformatik **Zentrum für Bioinformatik**

Programmierung für Naturwissenschaften 2
Sommersemester 2020
Übungen zur Vorlesung: Ausgabe am 30.04.2020

Zur Lösung der Aufgaben benötigen Sie einen C-Compiler (clang oder gcc), einen Texteditor und Enwicklungswerkzeuge wie z.B. `make`. Durch Aufruf von `clang --version` oder `gcc --version` können Sie überprüfen, ob die entsprechenden Programme installiert und die Pfade in Ihrer Pfadliste sind. Falls das nicht der Fall ist, müssen Sie evtl. die Variable `PATH` erweitern (siehe Beispiel in `bin/add_to_bash.sh`) oder die Programme nachträglich installieren. Das ist je nach Linux-Version unterschiedlich, aber mit einer Internetrecherche sind leicht Hinweise zu finden, wie man hier vorgeht. Unter macOS benötigen Sie `Xcode` und zur Verwaltung der nachträglichen installierten Pakete empfehle ich `Homebrew` `https://brew.sh`.

Bitte beachten Sie bei der Abgabe Ihrer Lösungen die Regeln, die im Dokument `doc/exercises_rules.pdf` beschrieben sind. Die Teilnehmer aus PfN1 kennen diese Regeln schon. Da immer wieder Fehler bei der Benennung der abzugebenden Dateien gemacht werden, habe ich ein Skript `bin/tar_loesungen.py` entwickelt, das das tar-file mit dem richtigen Namen erzeugt bzw. einen Fehler meldet, wenn das Verzeichnis mit Ihren Lösungen nicht korrekt benannt ist. Zur Verwendung des Skriptes müssen Sie lediglich eine Datei `namen.txt` anlegen, die zeilenweise in lexikographischer Reihenfolge die Nachnamen der Mitglieder Ihrer Gruppe angibt. Diese Datei muss nur einmal angelegt werden. Ein Änderung ist nur erforderlich, wenn sich die Gruppenzusammensetzung ändert.

Die hier genannten Pfade beziehen sich immer auf das Repository `pfn2_2020`. Um die Skripte aus dem `bin`-Verzeichnis einfache aufrufen zu können, muss dieses Verzeichnis in Ihrer Pfadliste sein. Ein Hinweis, wie man es zur Pfadliste hinzufügt, steht in `doc/pfn2_orga.pdf`. Die Hinweise beziehen sich auf

Aufgabe 1.1 (3 Punkte)

Implementieren Sie im Verzeichnis für diese Aufgabe ein C-Programm `hello.c`. Das Programm wird durch den Aufruf von `make` in diesem Verzeichnis kompiliert und, wenn alles gut geht, entsteht ein ausführbares Programm `hello.x`.

Je nach Aufruf des Programms von der Kommandozeile soll das Programm unterschiedliche Ausgaben liefern:

- Wenn kein Kommandozeilenparameter angegeben wird, dann soll `Hello World` ausgegeben werden.
- Sonst sollen alle Personen, mit den auf der Kommandozeile angegebenen Namen gegrüßt werden. `./hello.x Tim Struppi` soll dann z.B. die Ausgabe

  ```
  Hello Tim
  Hello Struppi
  ```

  liefern.

In dieser Aufgabe müssen Sie auf die Parameter `argc` und `argv` der Funktion `main` zugreifen, siehe Folie 26 in `C_slides.pdf`. Dabei ist `argv[0]` der Pfad des ausgeführten Programms, `argv[1]`

zeigt auf den String mit dem ersten Kommandozeilenparameter (falls vorhanden), `argv[2]` zeigt auf den String mit dem zweiten Kommandozeilenparameter (falls vorhanden), etc. Ein String wird mit Hilfe der Formatspezifikation `%s` im ersten Argument eines Aufrufs von `printf` ausgegeben.

Durch den Aufruf von `make test` im Verzeichnis zu dieser Aufgabe verifizieren Sie, dass Ihr Programm für einige Beispielaufrufe korrekt funktioniert.

Schauen Sie sich die Datei `Makefile` genauer an und kommentieren Sie, jeweils für jeden Abschnitt, was dieser bedeutet. Mit Abschnitt ist hier eine maximale Folge von Zeilen, die keine Leerzeilen enthalten, gemeint. Ihren Kommentarzeilen muss das Zeichen # vorangestellt werden. Es reichen jeweils pro Abschnitt 1-2 Zeilen mit Kommentaren.

Aufgabe 1.2 (3 Punkte)

Implementieren Sie im Verzeichnis für diese Aufgabe ein C-Programm `limits_tsv.c`. Das Programm wird durch den Aufruf von `make` in diesem Verzeichnis kompiliert und, wenn alles gut geht, entsteht ein ausführbares Programm `limits_tsv.x`.

Das Programm soll Größe, Minimal- und Maximalwerte von Variablen der Typen **`char`**, **`short`**, **`int`** und **`long`** jeweils in der **`signed`**- und **`unsigned`**-Variante ausgeben, und zwar in einem Tabulator-separierten Format mit einer Zeile pro Typ. Das genaue Format ergibt sich aus der Referenzdatei `limits.tsv` in den Materialien.

Da es sich hier um eine Ausgabe handelt, in der für jeden Typ die entsprechende Information im gleichen Format ausgegeben wird, kann man sich einige Tipparbeit ersparen, wenn man Makros verwendet, sinnvollerweise eins für die **`signed`**-Typen und eins für die **`unsigned`**-Typen. Die Makros erhalten als Parameter den Typ und den Maximalwert (bei **`signed`**-Typen auch den Minimalwert) und definieren die entsprechende `printf`-Anweisung. Dabei können Sie die Eigenschaft ausnutzen, dass ein Parameter, z.B. `TYPE` aus dem Kopf des Makros (vor dem ersten Leerzeichen) in der Definition des Makros (nach dem ersten Leerzeichen) durch Voranstellen des Symbols # zu einem String-Literal wird. D.h., wenn Sie `TYPE` mit **`int`** instantiieren wird der Präprozessor **`#`**`TYPE` durch `"int"` ersetzen. Ein Makro kann man auch über mehrere Zeilen schreiben, indem man die Zeilen durch ein \ trennt.

Bevor Sie eine Implementierung mit Makros beginnen, sollten Sie für die Typen **`long`** und **`unsigned long`** die `printf`-Anweisungen für das gewünschte Format implementieren und mit dem erwarteten Ergebnis vergleichen. Danach können Sie hiervon abstrahieren, um die Makros zu definieren, und diese dann für die anderen Typen verwenden. Wenn Sie sich noch nicht an Makros herantrauen, können Sie natürlich auch eine Lösung ohne Makros entwickeln, die wesentliche mehr Tipparbeit erfordert.

Hinweis: In der Datei `limits.h` (typischerweise im Verzeichnis `/usr/include`) finden Sie die Definitionen von (parameterlosen) Präprozessor-Makros für die Minimal- und Maximal-Werte der einzelnen Typen. So ist `INT_MAX` bzw. `UINT_MAX` das Makro für den größten Wert, den eine Variable vom Typ **`int`** bzw. **`unsigned int`** speichern kann. Die anderen Präprozessor-Makros sind analog definiert, außer für den Typ **`short`**, für den die Präprozessor-Makros mit `SHRT` beginnen.

Durch den Aufruf von `make test` im Verzeichnis zu dieser Aufgabe verifizieren Sie, dass Ihr Programm korrekt funktioniert.

Aufgabe 1.3 (4 Punkte) Implementieren Sie ein C-Programm `sumup.c`, das beim Aufruf über die Kommandozeile (also via `argv`) zwei nicht negative ganze Zahlen m und n erhält und die folgende

Summe berechnet und ausgibt:

$$\sum_{i=m}^{n} i$$

Falls $m > n$, dann ist diese Summe 0. Das Programm soll in einer Zeile durch Tabulatoren separiert die Werte von m und n sowie die Summe ausgeben (siehe Datei `run_result.tsv`).

Beispiel: Der Aufruf `./sumup.x 975159 1948264` liefert die Ausgabe

```
975159 1948264 1422400230919
```

Beachten Sie, dass bei der Berechnung wie in diesem Beispiel große Zahlen auftreten können. Daher müssen Sie in Ihrem Programm zur Berechnung der Summe einen ganzzahligen Typ verwenden, der große Zahlen repräsentieren kann. Da die Summen nicht negativ werden, verwenden Sie einen vorzeichenlosen Typ.

Verwenden Sie `sscanf` zur Konvertierung der Strings aus `argv[1]` und `argv[2]` in ganzzahlige Werte. Sie können sich an dem Beispiel auf Seite 91 in `C_slides.pdf` orientieren.

Berechnen Sie für den Fall $m \leq n$ die obige Summe auf zwei Weisen:

1. durch $m - n + 1$ Additionen der Werte zwischen m und n und
2. durch einen arithmetischen Ausdruck unter Nutzung der Eigenschaft

$$\sum_{i=0}^{k} i = \frac{k \cdot (k+1)}{2}, \tag{1}$$

   die für alle $k \geq 0$ gilt.

Für Berechnung in 2. müssen Sie sich überlegen, worin der Unterschied der Summierungen $\sum_{i=m}^{n} i$ und $\sum_{i=0}^{k} i$ besteht. Dann können Sie zweimal Gleichung (1) für verschiedene Werte von k anwenden und $\sum_{i=m}^{n} i$ mit einer konstanten Anzahl von Multiplikations-, Divisions- und Subtraktions-Operationen berechnen. Konstant heißt hier, dass die Anzahl der Operationen nicht von den Eingabewerten m und n abhängt.

Hinweis zum Infix-Operator `/` in C. Wenn beide Operanden ganze Zahlen sind, ist das Ergebnis wieder eine ganze Zahl. D.h. für zwei ganzzahlige Variablen a und b liefert a/b den Wert von $\frac{a}{b}$ ohne Rest. Falls $a = 111$ und $b = 2$, dann liefert a/b den Wert `55`.

Testen Sie in Ihrem Programm durch eine `assert`-Anweisung, dass die Ergebnisse der beiden Berechnungen aus 1. und 2. identisch sind.

Durch den Aufruf von `make test` im Verzeichnis zu dieser Aufgabe verifizieren Sie, dass Ihr Programm korrekt funktioniert.

Punktevergabe:

- 1 Punkt für das korrekte Einlesen
- 1 Punkt für die Berechnung mit $n - m + 1$ Additionen
- 1 Punkt für die Berechnung unter Verwendung der Summationsgleichung (1)
- 1 Punkt für bestandene Tests

Bitte die Lösungen zu diesen Aufgaben bis zum 05.05.2020 um 10:00 Uhr an pfn2@zbh.uni-hamburg.de schicken. Es erfolgt keine Besprechung der Lösungen.