Technische Universität München
Institut für Informatik I-16
Dr. Tobias Lasser
Jakob Vogel

WiSe 2012/13
Lösungsblatt 2
29. Oktober 2012

Algorithmen und Datenstrukturen

Aufgabe 1 Zahldarstellung (Beispiellösung)

a) Sinnvollerweise werden mindestens Ganzzahlentypen von 8, 16, 32 und 64 Bit Länge vorhanden sein, jeweils in einer `signed` und `unsigned` Variante.

b) Wir wählen etwa die Zahl $26_{10} = 0001\ 1010_2$ für unsere Berechnung. Die erste Verschiebung nach rechts ergibt

$$0001\ 1010_2 \rightsquigarrow 0000\ 1101_2 = 13_{10}\ ,$$

die zweite

$$0000\ 1101_2 \rightsquigarrow 0000\ 0110_2 = 6_{10}$$

und die dritte

$$0000\ 0110_2 \rightsquigarrow 0000\ 0011_2 = 3_{10}\ .$$

Offensichtlich ist also eine einzelne Verschiebung nach rechts die ganzzahlige Teilung durch 2, wobei der Rest nach rechts „herausgeschoben“ wird. Allgemein formuliert ist eine Verschiebung um n Stellen nach rechts eine Teilung durch 2^n.

Umgekehrt ergibt dann eine erste Verschiebung von 26_{10} nach links

$$0001\ 1010_2 \rightsquigarrow 0011\ 0100_2 = 52_{10}\ ,$$

eine zweite

$$0011\ 0100_2 \rightsquigarrow 0110\ 1000_2 = 104_{10}$$

und die dritte schließlich

$$0110\ 1000_2 \rightsquigarrow 1101\ 0000_2 = 208_{10}\ .$$

Der Trick funktioniert also auch in die andere Richtung, und eine einzelne Verschiebung nach links entspricht einer Multiplikation mit 2. Eine Verschiebung um n Stellen nach links ist also eine Multiplikation mit 2^n.

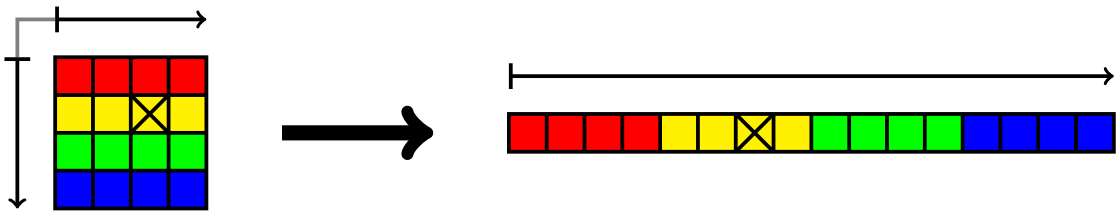
c) Für diese Aufgabe wählen wir beispielsweise die Zahl $99_{10} = 0110\ 0011_2$. Die Darstellung von 31_{10} in der Binärbasis ist $0001\ 1111_2$. Die Berechnung des bitweisen UND ergibt also:

$$\begin{array}{rl} & 0110\ 0011_2 \\ \wedge & 0001\ 1111_2 \\ \hline & 0000\ 0011_2 \end{array} = 3_{10}$$

Dies ist aber genau das Ergebnis von 99_{10} mod 32_{10}!

Allgemein kann man also sehr leicht den Divisionsrest für Zweierpotenzen $(2_{10})^n = 10...0_2$ berechnen, indem man mit $(2_{10})^n - 1_{10} = 01...1_2$ bitweise „ver-UND-et“.

d) Vergegenwärtigen Sie sich zunächst nochmals die Anordnung eines Bildes im Speicher, wobei das eigentlich zwei-dimensionale Bild zeilenweise hintereinander, also „linearisiert“ abgelegt wird:



In diesem Beispiel der Größe 4×4 betrachten wir nun das markierte gelbe Pixel (*Picture Element*). Wie bei Computern gewohnt beginnt die Zählung bei 0, und der horizontale Index wird vor dem vertikalen notiert. Die Markierung liegt also bei Index $(2/1)$.

Der Speicher im Rechner ist nun aber nur ein-dimensional organisiert. Daher werden die zwei- auf eindimensionale Indices abgebildet durch die bereits in der Angabe aufgeführte Formel $x + y \cdot \mathit{breite}$, hier also $2 + 1 \cdot 4 = 6$. (Der erste Kasten hat den Index 0; der Index 6 bezeichnet also den 7. Kasten!) Diese Umrechnung muss bei jedem Zugriff auf ein Pixel erfolgen!

Texturen in Computerspielen (also sozusagen die Bilder, die auf die Polygone zur Erzielung besseren Aussehens aufgemalt werden) werden auch so im Speicher hinterlegt. In einer komplexen virtuellen Welt wird bei der Darstellung dementsprechend sehr häufig diese Index-Berechnung ausgeführt. Wenn die Textur aber eine Größe von Zweierpotenzen hat, sagen wir $2^m \times 2^n$, dann kann die Multiplikation wie in der vorhergehenden Aufgabe durch einen (vor allem auf alten Prozessoren erheblich schnelleren und sehr gut in Hardware implementierbaren) Bitshift ersetzt werden, und der Index berechnet sich als `x + y << m`.

Außerdem werden bei der 3D-Modellierung gerne gekachelte Texturen angewendet. Eine große Ziegelmauer bekommt also ein Bild von nur ein paar wenigen Ziegeln zugewiesen, das dann über die ganze Mauer immer wieder wiederholt wird. Dazu müssen die Bildkoordinaten vor der Index-Berechnung erst in das kleine Bild umgerechnet (man spricht von „wrapping“) werden. Mathematisch ist dies wieder ein Rest nach Teilung, der im Fall von Zweierpotenzen – wie oben – durch ein schnelles bitweises UND implementiert werden kann.

Offensichtlich also erlaubt die Textur-Skalierung anhand von Zweierpotenzen schnelle und effiziente Implementierung[1].

e) Betrachten wir zuerst nochmals das aus der Vorlesung bekannte *Zweier-Komplement*. Eine gegebene Zahl, sagen wir $23_{10} = 0001\ 0111_2$, wird zunächst bitweise invertiert, anschließend wird 1 addiert:

$$\begin{array}{rcr} \neg\ 0001\ 0111_2 & = & 1110\ 1000_2 \\ & + & 0000\ 0001_2 \\ \hline & & 1110\ 1001_2 \end{array}$$

Diese Koeffizientenfolge wird bei einem `unsigned`-Typen interpretiert als $233_{10} = 256_{10} - 23_{10}$, bei einem `signed`-Typen dagegen als -23_{10}. Bei 8 Bit sind die darstellbaren Bereiche also 0...255 und $-128...127$ für die beiden Varianten. Interessanterweise können aber Variablen beider Arten ohne Konversion miteinander verrechnet werden, im Rahmen der Tatsache, dass Überläufe vorkommen können.

Daraus ergibt sich nun, dass negative Ganzzahlen im höchstwertigen Bit immer eine 1 gesetzt haben müssen. Eine negative 8-Bit-Zahl n kann also leicht erkannt werden durch den Test:

$$n \wedge 1000\ 0000_2 \stackrel{?}{=} 0000\ 0001_2$$

Die Berechnung des Absolutbetrages ist dagegen schon aufwändiger. Im Kern muss hier im Falle einer negativen Zahl das Zweier-Komplement berechnet werden. Betrachten wir also

[1] Wir vernachlässigen hier auch noch die sogenannten Mip-Map-Pyramiden, mit denen Aliasing-Effekte bei der Texturierung verringert werden. Deren Berechnung nutzt ebenfalls die Skalierung nach Zweier-Potenzen.

nochmals $-23_{10} = 1110\ 1001_2$ und hilfsweise die Maske $1111\ 1111_2$:

$$\begin{array}{rl} & 1110\ 1001_2 \\ + & 1111\ 1111_2 \\ \hline & 1110\ 1000_2 \\ \oplus & 1111\ 1111_2 \\ \hline & 0001\ 0111_2 \end{array} \quad = \quad 23_{10}$$

Im Falle einer negativen Zahl kann also durch Addition der Maske $1111\ 1111_2$ und anschließendes „ver-XOR-en“ der Betrag berechnet werden. Äquivalent dazu ist im Übrigen, zuerst das XOR durchzuführen, und die Maske anschließend zu subtrahieren. Führt man diese Operationen mit der Maske $0000\ 0000_2$ durch, bleibt der Wert dagegen unverändert. Bleibt also noch die Wahl der jeweils richtigen Maske für positive und negative Zahlen. Beachtet man die Tatsache, dass der Bitshift nach rechts bei **negativen** `signed`-Werten von links mit 1 statt mit 0 auffüllt[2], lautet der Code dann:

```
uint8_t absolutbetrag(int8_t zahl)
{
    int8_t maske = zahl >> 7;
    return (zahl + maske) ^ maske;
//  return (zahl ^ maske) - maske;
}
```

Es gibt eine ganze Reihe weiterer Funktionen, die durch bitweise Operationen optimiert durchgeführt werden können. Bei Interesse besuchen Sie bitte etwa die Seite `http://graphics.stanford.edu/~seander/bithacks.html`! Normalerweise benutzen Sie aber bitte die von der Programmiersprache vorgesehenen Operatoren und Funktionen.

Die auf dem Angabenblatt vorgegebene Schleife wird im Übrigen nie terminieren, da die Schleifenbedingung trivialerweise immer erfüllt ist – `unsigned`-Typen sind per Konstruktion immer ≥ 0!

f) Die zweite wichtige Klasse von primitiven Datentypen sind neben den Ganzzahlen die Gleitkommazahlen (*Floating Point*). Wie Sie in der Vorlesung gesehen haben, wird die Zahl in einer der „wissenschaftlichen Notation“ $\pm x \cdot 10^y$ ähnlichen Art gespeichert. Dies macht Rechenoperationen auf derartigen Zahlen ungleich komplexer, da normalerweise bei zwei Operanden erst die Exponenten angeglichen werden müssen, bevor Mantisse und Vorzeichen verrechnet werden können. Die Berechnung auf Fließkommazahlen erfordert also erheblichen Aufwand[3].

Die drei Komponenten der Fließkommazahl stehen direkt hintereinander im Speicher. Schiebeoperationen bringen die dadurch erforderliche Ordnung durcheinander. Auch andere Operationen erfordern Kenntnis von der genauen Struktur des Datenformats, werden gewöhnlich also vermieden.

Aufgabe 2 Arrays (Beispiellösung)

Die Lösung steht primär als Quellcode zur Verfügung, den Sie über die Website beziehen können. Dieser Text umfasst lediglich einige ergänzende Anmerkungen.

[2] Diese Eigenschaft wurde auf der Angabe noch nicht erwähnt, da dort nur `unsigned`-Typen verwendet wurden. Sie garantiert, dass auch bei Vorzeichen eine Division durch 2 durch den Rechts-Shift korrekt wiedergegeben wird.

[3] Dies wurde früher von einem eigenen Baustein, dem mathematischen Koprozessor erledigt, der heute in die CPU integriert wird. Vor allem eingebettete Systeme haben bis heute (oder hatten bis vor kurzem) wegen des Aufwands auch keine Unterstützung für Gleitkommazahlen. In diesem Fall behalf man sich etwa mit Fixed-Point-Typen.

Bedenken Sie zunächst, dass bei Arrays stets der **erste Index** 0 ist. Diese Zählung ist in der Informatik weithin üblich, und findet entsprechend auch bei mehrdimensionalen Datensätzen Verwendung.

In der Vorlesung haben Sie außerdem zwei weitere Methoden kennengelernt, ein Array zu erzeugen. Für C wurde folgende (auch unter C++ funktionierende) Schreibweise vorgestellt:

```
// Erzeuge ein Array der Laenge 5
int ein_array[5];
ein_array[0] = ...;
```

Hier ist zu beachten, dass die Länge 5 bereits zur Kompliationszeit feststehen muss. Es ist also **nicht erlaubt**, statt der 5 eine Variable anzugeben.

Die zweite Alternative ist unter C++ die Klasse `std::vector`, die Teil der Standardbibliothek ist. Um diese Möglichkeit zu nutzen, importieren Sie sie mittels `#include <vector>` zu Beginn Ihres Quellcodes. Die Klasse kapselt die Speicherverwaltung (verwendet dort aber auch `new[]` und `delete[]`), nimmt ihnen also viel Arbeit ab und gibt Ihnen dadurch sogar mehr Sicherheit.

```
// Erzeuge ein Array der Laenge 5
std::vector<int> ein_array(5);
ein_array[0] = ...;
```

Hier können Sie dann natürlich variable Größen angeben. Bitte verwenden Sie diese Variante in Zukunft, nun da Sie eine Vorstellung von deren Funktionsweise haben!

a) Der Prozess des paarweisen Vertauschens führt dazu, dass der Inhalt einer Zelle quer durch das Array bewegt wird. Wichtig dabei ist hauptsächlich, dass über eine Hilfsvariable getauscht werden muss, da sonst – je nach Reihenfolge – einer der zu tauschenden Werte ganz entfällt, und der andere dupliziert wird. Achten Sie beim Programmieren grundsätzlich auf derartige Fragen!

   Wenn man diesen Prozess um eine äußere Schleife erweitert, wie im zweiten Teil der Frage erwähnt, und dann auch noch einen Vergleich hinzufügt, erhält man bereits einen einfachen Sortieralgorithmus – *Bubblesort*. Der Name kommt daher, dass der Zelleninhalt dann, wie etwa Kohlensäurebläschen in Mineralwasser, nach oben „blubbert“.

b) Die Verwendung von globalen Variablen ist in C zwar häufig, in C++ allerdings selten und auch wenig elegant. Dies hat damit zu tun, dass diese Variablen dann im ganzen Programm zur Verfügung stehen, keinerlei Zugriffskontrolle stattfindet, und insofern eigentlich kein Vertrauen bezüglich deren Inhalte bestehen kann.

   Wir verwenden derartige Variablen hier nur, um einen über verschiedene Funktionen hinweg synchronen Zustand zu schaffen, ohne gleich eine Unmenge an Funktionsparametern austauschen zu müssen. Dies kann auch anderweitig erreicht werden, wie Sie in unmittelbar nachfolgenden Aufgaben sehen werden. Bitte vermeiden Sie also globale Variablen weitestgehend, und kapseln Sie Ihre Informationen in Klassen.

Aufgabe 3 Objektorientierung I (Beispiellösung)

Die Lösung steht primär als Quellcode zur Verfügung, den Sie über die Website beziehen können. Dieser Text umfasst lediglich einige ergänzende Anmerkungen. Beachten Sie dazu bitte auch die Erklärungen auf den Folien.

Die Idee hinter Objektorientierung ist die Schaffung neuer zusammengesetzter Datentypen. Man nennt einen derartigen Typen (sozusagen den Bauplan) eine Klasse, eine Variable dieses Typen dann eine Instanz.

Eine Klasse verfügt einerseits über Variablen beliebigen Typs (primitiv, oder auch zusammengesetzt), die pro Instanz unabhängig existieren. Diese Variablen machen zusammen den Zustand der Instanz aus. Um diesen Sachverhalt kenntlich zu machen, spricht man auch gerne von *Member Variables*.

Dieser Zustand wird normalerweise nicht extern manipuliert, sondern über Funktionen der Klasse, den sogenannten *Member Functions*. Diese Funktionen entsprechen weitestgehend normalen Methoden, bekommen jedoch immer zusätzlich einen „versteckten“ Verweis auf eine konkrete Instanz als Funktionsparameter übergeben. Auf dieser speziellen Instanz führen die Funktionen dann ihre Berechnungen aus.

Bei der Planung bzw. Skizzierung entsprechender Software-Systeme setzt man gerne eine grafische „Sprache“ – hier eher im übertragenen Sinn zu verstehen – namens UML (*Unified Modeling Language*). Vereinfacht würde man die Beispielklasse `Kreis` etwa so notieren:

Kreis

`x : float`
`y : float`
`radius : float`

`berechneLaengeDesOrtsvektors ( ) : float` `verschiebeKreis ( dx : float , dy : float ) : void`

In einem komplexeren Entwurf könnte man dann derartige Skizzen für mehrere Klassen anfertigen und die Beziehungen zwischen ihnen kenntlich machen. Da das auch bei kleineren Entwürfen schnell komplex werden kann, beschränkt man sich gerne auf die relevanteren Aspekte.

Streng genommen ist der vorgeschlagene Code über `struct` im Übrigen eine Mischung aus C und C++. Structs existieren bereits in C, können dort aber nur Variablen enthalten. Das eigentliche Konstrukt in C++ wäre eigentlich `class`, das weitestgehend wie `struct` verwendet wird, darüber hinaus aber auch noch Sichtbarkeiten kennt. Letztere definieren, wer einzelne Variablen nutzen oder Funktionen aufrufen darf.

Wir ignorieren diese Möglichkeit bis auf weiteres und arbeiten ohne Sichtbarkeiten. Daher verwenden wir `struct`, das laut Definition nur eine `class` ist, bei der alle Variablen und Funktionen sichtbar – also `public` – sind. Die Verwendung von `struct` in C lernen Sie im Praktikum!