Numerische Datentypen

Simon Weidmann

08.05.2014

1 Ganzzahlige Typen

1.1 Generelles

Bei Datentypen muss man immer zwei elementare Eigenschaften unterscheiden: Zuerst gibt es den Wertebereich, zweitens die zulässigen Operationen auf dem Typ.
Als numerische Datentypen in C gibt es (bekannterweise) z.B. char, short, int, long, float und double. Hinter diesen Typen steht jedoch um einiges mehr als man zunächst denken mag. chars zum Beispiel nimmt eine besondere Rolle ein, weil sie nicht nur Zahlen, sondern auch Buchstaben darstellen. Da in C der fertige Typ des Strings komplett fehlt, werden solche durch char-Arrays dargestellt. Mehr dazu jedoch in einem anderen Vortrag.

1.2 Signed / Unsigned

Die ganzzahligen Typen können noch eine weitere Eigenschaften haben: Sie können signed oder unsigned sein (signed int, unsigned int): Dies weist darauf hin, ob im Binärwort für den Wert das erste Bit als Vorzeichenbit verwendet wird, oder nicht. Nehmen wir als an, ein Integer hat eine Wortlänge von 4 Byte, dann geht sein Definitionsbereich als „signed int“ von -2147483648 bist 2147483647. Als „unsigned int“ jedoch geht der Wertebereich von 0 – 4.294.967.295. Bei „signed“ wird das erste Datenbit als Vorzeichenbit gelesen, als „unsigned“ wird der Wert immer als postitiv interpretiert. Somit vergrößert oder verkleinert sich der Wertebereich zwar nicht, er verschiebt sich nur ins positive/ negative.
Dies kann beispielsweise nützlich sein, wenn man große Mengen zählen will, oder generell keine negativen Ergebniswerte erwartet und deshalb den negativen Zahlenbereich nicht benötigt.

1.3 Überlauf

Aber was passiert, wenn man aus diesem Zahlenbereich herauszählt? Bekommt man eine Fehlermeldung? Nein! Es findet ein sogenannter Überlauf statt. Das heißt, dass man, sobald man versucht an der oberen Grenze aus dem Zahlenbereich herauszusteigen, unten wieder anfängt (und andersherum). Es gilt also $MAX + 1 = MIN$, wobei MAX gleich der oberen Grenze des Wertebereichs und MIN gleich der unteren Grenze ist. Der Wertebereich eines numerischen Datentyps lässt sich in Folge dessen als ein Zahlenkreis darstellen (siehe Bild). Man muss beim „overflow“ jedoch eine wesentliche

Unterscheidung machen. Bei unsigned Datentypen gilt der overflow wie eben geschildert. Für signed Datentypen jedoch ist der overflow undefiniert. Gerade deshalb muss man sich immer vergewissern, dass man nicht in den signed overflow gerät.

$$Beispiel : ([0, 255]) \ \rightarrow \ 200 + 200 = 154 = 400 - 256$$

2 Gleitkommazahlen

Neben den ganzen Zahlen gibt es natürlich auch Kommazahlen. Diese beim Programmieren zu nutzen, scheint völlig selbstverständlich. Diese sind aber für den Computer, der nur auf Binärwörtern arbeiten kann, ein wenig speziell zu definieren. Gespeichert werden „float“ und „double“ im folgenden Stil:

$$Vorzeichen * Mantisse * 2^{Exponent}.$$

Diese Zahlen nennen sich Gleitkomma-, oder auch Fließkommazahlen, da der variable Exponent das Komma „gleiten“ oder auch „fließen“ lässt. „Gewöhnliche Dezimalzahlen“ binär darzustellen, ist schwierig und relativ ungenau. Zum Beispiel die Dezimalzahl 0.1: Man kann sie weder mit float, noch mit double genau darstellen, nur den ungefähren Wert. Denn der Computer berechnet nur auf diejenige Genauigkeit, die auf Basis der Größe des Datentyps möglich ist. Dafür wird heutzutage der „IEEE 754“ Standard verwendet. Aus diesem geht hervor, dass dem Exponenten bei „floats“ 8 Bit im Speicher zustehen, mit 23 Bit Mantisse, und bei „doubles“ 11 Bit für den Exponenten mit den überbleibenden 52 für die Mantisse. Wenn man mit exakten Werten rechnen möchte, sollte man also eher „double“ verwenden, um technisch basierte Rundungsfehler zu vermeiden. Denn „double“ hat, wie der Name schon impliziert (mehr als) doppelte Genauigkeit.

3 Typumwandlung

3.1 Typecasting

Implizites casten: Wenn man zwei verschiedene Typen versucht zu verrechnen, wird automatisch der kleinere Typ (Bitzahl) in den größeren Typen umgewandelt:

$$Bsp : int + short = int.$$

Bei einer Umwandlung von chars muss man besonders aufpassen, weil diese als unsigned interpretiert werden. Man kann Typen aber auch explizit casten. Zum Beispiel wird double d = 2.5 in einen float gecastet, indem man float f = (float) d schreibt. Dabei geht Genauigkeit verloren, man spart aber natürlich Speicher.

3.2 Literale

Wenn man nicht jeden numerischen Typen immer explizit casten möchte, kann man auch ganz einfach mit „Literalen“ arbeiten. So erhält man beispielsweise mit 100 einen Integer-Wert, mit 100l einen Long, mit 100ul einen Unsigned Long, mit 100f einen float, und so weiter. Je nach Postfix erhält man einen anderen Datentyp.
Zudem kann man auch Zahlen anstatt als Dezimalzahl als Hexadezimalzahl mit dem Präfix 0x, oder als Oktalzahl mit dem Präfix 0 schreiben. Es gibt also beispielsweise:

$$0xFF = 0377 = 255$$

Gerade, wenn man mit als Zahlen codierten Bitmustern (Stichwort Maskierung) arbeitet, können einem diese Präfix wirklich viel Arbeit zum Umrechnen ersparen.
Für eine Auflistung von weiteren Literalen verweise ich auf die Quelle von tutorialspoint.com.

4 Operationen

Es gibt mehrere Arten von binären Operatoren auf den numerischen Datentypen. Zunächst einmal die arithmetischen wie "‘+“ und „++“, die uns allen seit der Grundschule bekannt sind, also die rein mathematischen, wie Addition oder Inkrement $(i + 1)$.
Dann gibt es noch boolsche Operationen, die vergleichend sind und 1 oder 0 als Ergebnis haben. Da es nicht wie in Java den Typen „boolean“ gibt, interpretiert C alles ! = 0 als true, und 0 als false. Es folgt z.B. 1&&2 = 1. Weitere Beispiele für boolsche Operatoren sind „<“ oder „||“. Zuletzt gibt es noch bitweise Operationen, wie „&“ oder „|“. Diese sind nicht zu verwechseln mit den boolschen „&&“ und „||“. Bei diesen wird nämlich die Zahl, auf die man sich bezieht, bitweise der logischen Operation unterzogen. Wir wissen ja, dass jede Zahl intern als Bitmuster, als Binärzahl dargestellt wird. Beispielsweise ist $14 = 1110$ $(8 + 4 + 2)$. bei den bitweisen Operationen werden nun zwei dieser Wörter von rechts nach links logisch verknüpft. Exemplarisch

nehmen wir dafür „&“ als solche Verknüpfung. Das Ergebnis an einer Stelle im Wort ist genau dann 1, wenn beide verknüpften Wörter dort eine 1 haben. Zu berechnen sei nun 14 & 11. $14 = (1110)_2$ und $11 = (1011)_2$ Durch bitweise „verundung“ ergibt sich:

$$\begin{aligned} 1110\ \& \\ \underline{1011\ =} \\ 1010 \\ (1010)_2 = 10 \\ \Rightarrow 14\ \&\ 11 = 10 \end{aligned}$$

Zusammengefasst können also für die Operation „und“ in C drei verschiedene Ergebnisse entstehen:

$$\begin{aligned} 14 + 11 &= 25 \\ 14\ \&\&\ 11 &= 1 \\ 14\ \&\ 11 &= 10 \end{aligned}$$

Interessant ist nun auf jeden Fall noch die Präzedenz, also die Bindungsstärke der verschiedenen Operatoren und Operatorentypen. Dazu verweise ich auf die Quelle http://www.swansontec.com/sopc.html. Dort sind die verschiedenen Operatoren nach Bindungsstärke geordnet. Man sollte jedoch auf jeden Fall die elementarsten Operatoren ordnen können. Z. B. && vor ||, sowie & vor |.

5 Architekturabhängigkeiten

Je nach Betriebssystem haben die verschiedene Typen verschiedene Wortlängen. So kann ein normaler int 16 Byte oder auch 32 Byte groß sein.
Wenn man portabel programmieren möchte, ist das natürlich ein Problem. So gerät man sehr schnell in den ungewünschten und unbeliebten overflow, wenn man plötzlich nicht mehr mit 32 Byte Integern sondern mit 16 Byte pro Integer rechnet. Aber auch in der "pointer-Arithmetik"(späterer Vortrag) hat der pointer nicht mehr unbedingt die Länge eines ints, wenn man das Betriebssystem wechselt. Zum Glück gibt es die sogenannten „portable redefined types“ in der Bibliothek inttypes.h. Mit denen umgeht man das Problem einfach, indem man statt int hier einfach intmax_t verwendet. Der Ausdruck verwendet dann je nach Betriebssystem den längstmöglichen Integertypen. Das erspart viel Aufwand, um Programme portabel zu designen. Es gibt noch weitere von diesen redefined types, für 8, 16, 32, 64 und 128 Bits Wortlänge.