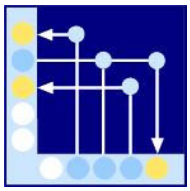

Fakultät Elektronik und Informatik
Studiengang Informatik



# Programmieren in C++

Vorlesung im Wintersemester 2017/2018
Prof. Dr. habil. Christian Heinlein

## 1. Übungsblatt (16. Oktober 2017)

## Aufgabe 1: Ganze Zahlen

### Teilaufgabe 1.a)

Finden Sie heraus, wie groß die Typen `short`, `int`, `long` und `long long` auf Ihrem System sind und ermitteln Sie ihre minimalen und maximalen Werte!
Was lässt sich damit über die interne Darstellung negativer Werte aussagen?

Finden Sie außerdem heraus, welche Typen sich hinter den Aliasen `int_least16_t` und `int_fast16_t` verbergen!
Tipp: Übergeben Sie Werte dieser Typen jeweils an eine Funktion `test`, die für alle in Frage kommenden Typen überladen ist!

### Teilaufgabe 1.b)

Überprüfen Sie die in §2.1.4 der Vorlesungsfolien genannten Regeln, nach denen der Typ eines ganzzahligen Literals ermittelt wird, indem sie gezielt dezimale, oktale, hexadezimale und duale Literale mit bestimmten Werten erstellen und ihren Typ ermitteln!

### Teilaufgabe 1.c)

Überprüfen Sie die in §2.1.8 genannten Regeln, nach denen bei einer arithmetischen Operation der gemeinsame Typ der Operanden – der gleichzeitig auch der Resultattyp der Operation ist – ermittelt wird!

Wie lautet demnach das Ergebnis von `1U - 2U` und das von `(unsigned short)1 - (unsigned short)2`?

### Teilaufgabe 1.d)

Für welche `int`-Werte `x` und `y` liegt `x/y` nicht im Wertebereich von `int`, sofern `int`-Werte im Zweierkomplement dargestellt werden?

✎

```
#include <iostream>
#include <limits>
using namespace std;

// Makro, um Schreibarbeit zu sparen und Codeverdopplung zu vermeiden.
// (Alternativ könnte man test als Template-Funktion definieren.)
// #T wird vom Präprozessor durch den Parameter T als Zeichenkette
// ersetzt.
#define TEST(T) \
    void test (T x) { \
        cout << #T << " " << x << " (" << sizeof(T) << " bytes) [" \
            << numeric_limits<T>::min() << ", " \
            << numeric_limits<T>::max() << "]" << endl; \
    }

// Definition der Funktion test mit verschiedenen Parametertypen.
TEST(short)
TEST(int)
TEST(unsigned int)
TEST(long)
TEST(unsigned long)
TEST(long long)
TEST(unsigned long long)
TEST(double)

int main () {
    // Aufrufe von test für Werte der Typen short, int, long
    // und long long.
    test(short(1));
    test(1);
    test(1L);
    test(1LL);

    // Aufrufe von test für Werte der Typen int_least16_t
    // und int_fast16_t.
    test(int_least16_t(1));
    test(int_fast16_t(1));

    // Aufrufe von test für unterschiedliche ganzzahlige Literale.
    // Annahme: int hat 32 Bit, long und long long haben 64 Bit.
    // Dann kann int Werte kleiner als 2 hoch 31 darstellen,
    // unsigned int Werte kleiner als 2 hoch 32.
    // 2 hoch 31 ist etwas größer als 2'000'000'000,
    // 2 hoch 32 ist etwas größer als 4'000'000'000.

    // Für dezimale Literale ohne Suffix u oder U
    // werden keine vorzeichenlosen Typen verwendet.
    test(2'000'000'000);        // int
    test(4'000'000'000);        // long
    test(8'000'000'000);        // long
```

```
    // Für oktale, hexadezimale und duale Literale ggf. schon.
    test(   0B100'0000'0000'0000'0000'0000'0000'0000);  // int
    test(  0B1000'0000'0000'0000'0000'0000'0000'0000);  // unsigned int
    test(0B1'0000'0000'0000'0000'0000'0000'0000'0000);  // long

    // Aufrufe von test für das Ergebnis arithmetischer Operationen
    // mit unterschiedlichen Operandentypen.
    test(short(1) + 1U);        // short + unsigned int -> unsigned int
    test(1 + 1U);               // int + unsigned int -> unsigned int
    test('x' + 1.0);            // char + double -> double

    cout << 1U - 2U << endl;    // -1 mod (2 hoch 32) = 4294967295
    cout << (unsigned short)1 - (unsigned short)2 << endl; // -1
}
```

Wenn der Wertebereich eines ganzzahligen Typs asymmetrisch ist (z. B. `-32768` bis `32767` bei `short`), dann werden negative Werte offenbar im Zweierkomplement dargestellt.

Auf einem 64-Bit-PC ist `int_least16_t` normalerweise gleich `short` (16 Bit), aber `int_fast16_t` möglicherweise gleich `long` (64 Bit), weil sich 64-Bit-Werte dort schneller verarbeiten lassen als 16-Bit-Werte.

Für `y = -1` ist `x/y` normalerweise gleich `-x`.
Wenn `int`-Werte im Zweierkomplement dargestellt werden, liegt dieser Wert für `x = numeric_limits<int>::min()` jedoch nicht im Wertebereich von `int`.

## Aufgabe 2: Lange vorzeichenlose ganze Zahlen

Implementieren Sie Addition und Multiplikation langer vorzeichenloser ganzer Zahlen!

Repräsentieren Sie eine solche Zahl quasi zur Basis 256 (allgemeiner `numeric_limits<unsigned char>::max() + 1` oder `(unsigned char)-1`), indem Sie sie als dynamisches Feld von `unsigned-char`-Werten speichern, dessen erstes Element als Längenfeld dient! Realisieren Sie die Operationen dann ähnlich wie beim schriftlichen Addieren und Multiplizieren!
Verwenden Sie für Zwischenergebnisse einen größeren ganzzahligen Typ, beispielsweise `uint_fast32_t`, damit Überträge nicht verloren gehen! Außerdem können Sie ausnutzen, dass arithmetische Operationen mit vorzeichenlosen Werten der Größe $n$ Bit immer korrekt modulo $2^n$ sind!

Schreiben Sie außerdem Funktionen zum Erstellen solcher Zahlen aus „kleinen“ `unsigned-char`-Werten sowie zur hexadezimalen Ausgabe solcher Zahlen!

```
#include <cstdint>
#include <iostream>
#include <iomanip>
#include <limits>
#include <string>
using namespace std;
```

```
// Ein bigint-Wert bi ist ein dynamisches Feld von unsigned-char-Werten,
// dessen erstes Element als Längenfeld dient.
// Das heißt: Wenn bi[0] gleich n ist, enthalten bi[1] bis bi[n] die
// Stellen der langen Zahl zur Basis B.
// Der Wert der Zahl ist dann:
// bi[n] * B hoch (n-1) + ... + bi[1] * B hoch 0.
typedef unsigned char* bigint;
const uint_fast32_t B = numeric_limits<unsigned char>::max() + 1;

// Lange Zahl mit n Stellen beschaffen.
bigint bi_new (int n) {
    bigint z = new unsigned char [n + 1];
    z[0] = n;
    return z;
}

// Speicher der langen Zahl x freigeben.
void bi_del (bigint x) {
    delete [] x;
}

// Größe der langen Zahl x als L-Wert liefern.
unsigned char& bi_size (bigint x) {
    return x[0];
}

// Einstellige lange Zahl mit Wert x erzeugen.
bigint bi_make (unsigned char x) {
    bigint z = bi_new(1);
    z[1] = x;
    return z;
}

// Lange Zahl x hexadezimal ausgeben.
void bi_print (bigint x) {
    // Ausgabestrom cout auf hexadezimale Ausgabe umstellen.
    cout << hex;

    // Stellen von x jeweils mit Breite 2 und ggf. führender Null
    // ausgeben.
    // Damit x[i] nicht als Zeichen, sondern als Zahl ausgegeben wird,
    // muss es in einen ganzzahligen Typ ungleich char, signed char
    // oder unsigned char umgewandelt werden.
    // Zwischen den Stellen wird jeweils ein Leerzeichen ausgegeben.
    for (int i = bi_size(x); i >= 1; i--) {
        cout << setw(2) << setfill('0') << (unsigned int)x[i];
        if (i > 1) cout << " ";
    }

    // Ausgabestrom cout wieder auf dezimale Ausgabe umstellen
    // und einen abschließenden Zeilentrenner ausgeben.
    cout << dec << endl;
}
```

```
// Maximum von m und n liefern.
int max (int m, int n) {
    return m > n ? m : n;
}

// Lange Zahlen x und y addieren.
bigint bi_add (bigint x, bigint y) {
    // Das Ergebnis z hat mindestens so viele Stellen wie x und y,
    // wegen Übertrag eventuell noch eine mehr.
    int n = max(bi_size(x), bi_size(y));
    bigint z = bi_new(n + 1);

    // Ergebnis z stellenweise berechnen.
    // Bei der Zuweisung an z[i] wird automatisch modulo B gerechnet.
    // Ein eventueller Übertrag verbleibt jeweils in tmp.
    uint_fast32_t tmp = 0;
    for (int i = 1; i <= n; i++) {
        if (i <= bi_size(x)) tmp += x[i];
        if (i <= bi_size(y)) tmp += y[i];
        z[i] = tmp;
        tmp /= B;
    }

    // Wenn am Ende ein Übertrag verbleibt,
    // wird er als Stelle n+1 gespeichert.
    // Andernfalls hat das Ergebnis nur n Stellen.
    // (z[n+1] bleibt dann unbenutzt.)
    if (tmp) z[n+1] = tmp;
    else bi_size(z)--;

    return z;
}

// Lange Zahlen x und y multiplizieren.
bigint bi_mul (bigint x, bigint y) {
    // Das Ergebnis z hat höchstens so viele Stellen wie x und y
    // zusammen, eventuell eine weniger.
    int n = bi_size(x) + bi_size(y);
    bigint z = bi_new(n);
```

```
    // Ergebnis z stellenweise berechnen.
    // z[i] ist die Summe aller Produkte x[j] * y[k]
    // mit (j-1) + (k-1) = (i-1), d. h. k = i - j + 1,
    // sofern die Stellen x[j] und y[k] existieren.
    // z[1] ist z. B. gleich x[1] * y[1],
    // z[2] gleich x[1] * y[2] + x[2] * y[1].
    // Ein eventueller Übertrag verbleibt jeweils in tmp,
    // das hierfür ausreichend groß sein muss.
    uint_fast32_t tmp = 0;
    for (int i = 1; i < n; i++) {
        for (int j = 1; j <= i; j++) {
            int k = i - j + 1;
            if (j <= bi_size(x) && k <= bi_size(y)) {
                tmp += x[j] * y[k];
            }
        }
        z[i] = tmp;
        tmp /= B;
    }

    // Wenn am Ende ein Übertrag verbleibt,
    // wird er als Stelle n gespeichert.
    // Andernfalls hat das Ergebnis nur n-1 Stellen.
    // (z[n] bleibt dann unbenutzt.)
    if (tmp) z[n] = tmp;
    else bi_size(z)--;

    return z;
}

// Stack von langen Zahlen, der in main allokiert wird.
bigint* stack;

// Index des nächsten freien Stackelements.
int top = 0;

// Lange Zahl x auf den Stack legen.
void push (bigint x) {
    stack[top++] = x;
}

// Die oberste lange Zahl vom Stack holen.
bigint pop () {
    return stack[--top];
}

// Testprogramm.
// Die Kommandozeilenargumente werden als Ausdruck in umgekehrter
// polnischer Notation interpretiert, dessen Wert ausgegeben wird.
int main (int argc, char** argv) {
    // Stack mit garantiert ausreichender Größe allokieren.
    stack = new bigint [argc - 1];
```

```
    // Kommandozeilenargumente verarbeiten.
    for (int i = 1; i < argc; i++) {
        string s = argv[i];
        if (s == "+" || s == "*") {
            // Operanden y und x vom Stack holen,
            // entweder ihre Summe oder ihr Produkt auf den Stack legen
            // und ihren Speicher freigeben.
            bigint y = pop(), x = pop();
            push((s == "+" ? bi_add : bi_mul)(x, y));
            bi_del(x); bi_del(y);
        }
        else {
            // s in eine lange Zahl umwandeln und auf den Stack legen.
            push(bi_make(stoi(s)));
        }
    }

    // Die oberste lange Zahl vom Stack holen und ausgeben.
    bi_print(pop());
}
```