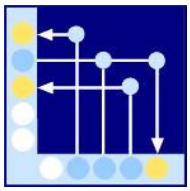
Fakultät Elektronik und Informatik
Studiengang Informatik



Programmieren in C++

Vorlesung im Sommersemester 2018
Prof. Dr. habil. Christian Heinlein

2. Übungsblatt (22. März 2018)

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 + 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());
}
```

✎

Aufgabe 3: Konstante Zeiger und Zeiger auf konstante Objekte

Gegeben seien die folgenden Deklarationen (vgl. §2.4.2):

```
// Gewöhnlicher Zeiger.
char* p;

// Zeiger auf konstantes Objekt.
const char* pc;
// Oder:
char const* pc;

// Konstanter Zeiger.
char* const cp = ...;

// Konstanter Zeiger auf konstantes Objekt.
const char* const cpc = ...;
// Oder:
char const* const cpc = ...;
```

Welche der folgenden Zuweisungen sind zulässig? Begründen Sie Ihre Antwort!

✎

l = r	p	pc	cp	cpc
p	+	−	+	−
pc	+	+	+	+
cp	−	−	−	−
cpc	−	−	−	−

Zuweisungen an `p`: Da `*p` nicht konstant ist, darf man keinen Zeiger `q` zuweisen, für den `*q` konstant ist.

Zuweisungen an `pc`: Da `*pc` konstant ist, darf man beliebige Zeiger `q` zuweisen.

Zuweisungen an `cp` und `cpc`: Da diese Variablen konstant sind, darf man grundsätzlich nicht zuweisen.
Zur *Initialisierung* von `cp` könnte man `p` (und `cp`) verwenden, zur Initialisierung von `cpc` jeden anderen Zeiger.

Aufgabe 4: Felder und Zeiger

Implementieren Sie die Bibliotheksfunktionen `strlen`, `strcpy`, `strcat` und `strdup`, die in der Definitionsdatei `<cstring>` definiert sind!

Hinweis: Je nachdem, ob man die Parameter der Funktionen direkt als Zeiger oder aber als Felder interpretiert bzw. verwendet, erhält man unterschiedlich kompakte Implementierungen.

Feldbasierte Implementierung

```
#include <cstdlib> // size_t

// Länge der Zeichenkette s bestimmen.
size_t strlen (const char* s) {
    int i = 0;
    while (s[i] != '\0') i = i + 1;
    return i;
}

// Zeichenkette src nach dest kopieren.
char* strcpy (char* dest, const char* src) {
    int i = 0;
    while (src[i] != '\0') {
        dest[i] = src[i];
        i = i + 1;
    }
    dest[i] = '\0';
    return dest;
}
```

```
// Zeichenkette src am Ende von dest anfügen.
char* strcat (char* dest, const char* src) {
    int i = 0, j = 0;
    while (dest[i] != '\0') i = i + 1;
    while (src[j] != '\0') {
        dest[i] = src[j];
        i = i + 1;
        j = j + 1;
    }
    dest[i] = '\0';
    return dest;
}

// Dynamische Kopie der Zeichenkette s erstellen.
char* strdup (const char* s) {
    int n = strlen(s) + 1;
    char* p = new char [n];
    return strcpy(p, s);
}
```

Zeigerbasierte (und kompaktierte) Implementierung

```
#include <cstdlib> // size_t

// Länge der Zeichenkette s bestimmen.
size_t strlen (const char* s) {
    const char* p = s;
    while (*s) s++;
    return s - p;
}

// Zeichenkette src nach dest kopieren.
char* strcpy (char* dest, const char* src) {
    char* p = dest;
    while (*dest++ = *src++) ;
    return p;
}

// Zeichenkette src am Ende von dest anfügen.
char* strcat (char* dest, const char* src) {
    char* p = dest;
    while (*dest) dest++;
    while (*dest++ = *src++) ;
    return p;
}

// Dynamische Kopie der Zeichenkette s erstellen.
char* strdup (const char* s) {
    return strcpy(new char [strlen(s) + 1], s);
}
```

Anmerkung

Wenn `new` keinen dynamischen Speicherplatz beschaffen kann, wird eine Ausnahme des Typs `bad_alloc` geworfen. In diesem Fall bricht `strdup` mit dieser Ausnahme ab, anstatt – wie im Handbuch beschrieben – einen Nullzeiger zurückzuliefern.