

**Einführung in die Programmiersprachen C und C++**

Prof. Dr. Ulf Rehmann, Fakultät für Mathematik

Übungsblatt 10 (4 Seiten)

**Beispiel für das Arbeiten mit einer Bibliothek****Die GNU-Multiple-Precision-Bibliothek GMP:**

Diese Bibliothek für Langzahlen-Arithmetik ist sehr effizient in C geschrieben, die Funktionen sind allerdings ziemlich unhandlich, es lohnt sich eine Erschliessung durch einen C++-“Wrapper”.

Hierbei sollten Effizienz-Fragen eine wesentliche Rolle spielen.

Hier wird keine fertige Lösung angeboten, sondern es werden einige Hinweise und Werkzeuge gegeben (siehe GMP.h).

Aktuelle Version der Bibliothek auf Ftp-Servern, z. B. in prep.ai.mit.edu unter /gnu/. (Einloggen als User “anonymous”).

Das folgende gilt sinngemäß für die meiste GNU- und sonstige Public Domain Software.

Datei: **gmp-3.1.tar.gz** oder auch **gmp-3.1.tgz**. Auspacken mit **tar xzf gmp-3.1.tar.gz** liefert ein Verzeichnis **gmp-3.1** mit Quellcode und Dokumentation (Unterverzeichnis **doc**) sowie eine Datei **README** mit weiteren Hinweisen und eine Datei **INSTALL** mit Anleitung zur Installation.

Meist ist das Ausführen der folgenden Befehle erforderlich:

```
./configure      (Erstellen der “Makefiles”)
make            (Compilieren des Codes)
make install    (Installieren im System für den ständigen Gebrauch in Unix).
```

Der letzte Befehl muss in der Regel mit **root**-Berechtigung erfolgen.

In der Datei **INSTALL** findet sich der Hinweis auf die “Info”-Unterlagen. Im Kapitel über **integers** findet man Hinweise auf die definierten Typen und vorhandenen Funktionen:

Variable sind z. B. vor Benutzung zu initialisieren:

```
mpz_t pie;
mpz_init_set_str (pie, "3141592653589793238462643383279502884", 10);
...
mpz_sub (pie, ...);
...
mpz_clear (pie);
```

```
- Function: void mpz_init_set (mpz_t ROP, mpz_t OP)
- Function: void mpz_init_set_ui (mpz_t ROP, unsigned long int OP)
- Function: void mpz_init_set_si (mpz_t ROP, signed long int OP)
- Function: void mpz_init_set_d (mpz_t ROP, double OP)
```

Als C++-Funktion kann man überladen. Der Typ **mpz\_t** ist eine “Referenz”-Version von **MP\_INT**, siehe die Datei **gmp.h**:

```
typedef unsigned long ulong;
typedef unsigned int uint;

inline void mpz_set (MP_INT *i, long a) { mpz_set_si(i,a); }
inline void mpz_set (MP_INT *i, int a) { mpz_set_si(i,a); }
inline void mpz_set (MP_INT *i, ulong a) { mpz_set_ui(i,a); }
inline void mpz_set (MP_INT *i, uint a) { mpz_set_ui(i,a); }
inline void mpz_set (MP_INT *i, double a) { mpz_set_d (i,a); }
inline void mpz_set (MP_INT *i, char *s) { mpz_set_str(i,s,10); }
```

Diese sollten in die C++-Konstruktoren aufgenommen werden. Zur elementaren Arithmetik gibt es:

```
- Function: void mpz_add (mpz_t ROP, mpz_t OP1, mpz_t OP2)
- Function: void mpz_add_ui (mpz_t ROP, mpz_t OP1, unsigned long int OP2)
```

Set ROP to  $OP1 + OP2$ .

```
- Function: void mpz_sub (mpz_t ROP, mpz_t OP1, mpz_t OP2)
- Function: void mpz_sub_ui (mpz_t ROP, mpz_t OP1, unsigned long int OP2)
  Set ROP to  $OP1 - OP2$ .

- Function: void mpz_mul (mpz_t ROP, mpz_t OP1, mpz_t OP2)
- Function: void mpz_mul_si (mpz_t ROP, mpz_t OP1, long int OP2)
- Function: void mpz_mul_ui (mpz_t ROP, mpz_t OP1, unsigned long int OP2)
  Set ROP to  $OP1 \text{ times } OP2$ .

- Function: void mpz_addmul_ui (mpz_t ROP, mpz_t OP1, unsigned long int OP2)
  Add  $OP1 \text{ times } OP2$  to ROP.

- Function: void mpz_mul_2exp (mpz_t ROP, mpz_t OP1, unsigned long int OP2)
  Set ROP to  $OP1 \text{ times } 2 \text{ raised to } OP2$ . This operation can also be
  defined as a left shift,  $OP2$  steps.

- Function: void mpz_neg (mpz_t ROP, mpz_t OP)
  Set ROP to  $-OP$ .

- Function: void mpz_abs (mpz_t ROP, mpz_t OP)
  Set ROP to the absolute value of OP.
```

Diese Funktionen sollten durch Überladen von C++-Operatoren erschlossen werden:

```
inline void mpz_add (MP_INT *z, MP_INT *x, ulong a){ mpz_add_ui(z,x,a); }
inline void mpz_add (MP_INT *z, MP_INT *x, long a){
  a>0 ? mpz_add_ui (z,x,a) : mpz_sub_ui (z,x,-a);
}
inline void mpz_add (MP_INT *z, MP_INT *x, uint a){ mpz_add_ui (z,x,a); }
inline void mpz_add (MP_INT *z, MP_INT *x, int a){ mpz_add (z,x,(long) a); }

...
```

```
// Addition
friend Int operator+(const Int& x, const Int& y) {
  Int z; mpz_add (z.p->i, x.p->i, y.p->i); return z;
}

template<class T> friend Int operator+(const Int& x, T y) {
  Int z; mpz_add (z.p->i, x.p->i, y); return z;
};

template<class T> friend Int operator+(T y, const Int& x) { return x+y; }
```

Die Langzahlen selbst sollten mittels Referenz-Zähler verwaltet werden, um Kopieraufwand zu vermeiden. Jedes Datenobjekt erhält eine Zähler-Komponente **n**:

```
class Int {
  struct Irep {
    MP_INT *i;
    int n;
    Irep() { n = 1; mpz_init(i = new MP_INT); } //Konstruktor
    ~Irep() { mpz_clear(i); delete i; } //Destruktor
  };
  Irep *p;
public:
```

Dieser Zähler führt Buch über die Anzahl der Referenzen, die auf dies Objekt zeigen. Auf diese Weise wird z. B. bei einer Zuweisung `Int x,y; ... y = x;` nicht das ganze Datenobjekt kopiert, sondern `y` zeigt lediglich auf das Datenobjekt von `x`, es muss also der Zähler des Datenobjektes, auf den `y` vor der Zuweisung zeigte, dekrementiert und der Zähler des Datenobjektes, auf den `x` zeigt, inkrementiert werden. Falls bei der Dekrementierung 0 erreicht wird, kann das Datenobjekt gelöscht werden, so dass der Bereich für andere Zwecke frei wird. Ferner muss der Zeiger von `x` nach `y` kopiert werden. Dies drei Operationen sind schneller als die evtl Kopie von einigen -zig Byte für die Zahl selbst. Hier der Zuweisungscode aus `GMP.h`:

```
Int operator=(const Int& x) {
    x.p->n++; if (--p->n == 0) { delete p; }
    p = x.p; return *this;
}
```

Ähnlich sind andere Operatoren zu behandeln.

Hier ein Anwendungsprogramm mit Timing: Die Laufzeit selbstdefinierter Funktionen (Fakultätsberechnung) wird mit der in `gmp` eingebauten Funktion "factorial" verglichen, um das Overhead durch Konstruktoren und Operatoren zu beurteilen:

```
// gmp-test.cc
// erfordert Gnu Multiple Precision Library (GMP)
//
// zu uebersetzen mit g++ gmp-test.cc -I/vol/gnu/include -lgmp
// Eine Test-Suite, in main werden einige der angegebenen
// Funktionen auf Laufzeit getestet
// andere wie fib sollten ebenfalls erprobt werden.
```

```
#include <stdio.h>
#include <iostream.h>
#include "Timer.h"
#include "GMP.h"
```

```
typedef long unsigned int ulong;
```

```
Int fib(long n) {
    Int ii, jj, kk;
    long int i;
    ii = 0; jj = 1; kk = 22;
    for (i=0; i < n; i++) {
        kk = ii+jj; ii = jj; jj = kk;
    }
    return ii;
}
```

```
Int fakultaet(ulong n) {
    return n <= 1 ? 1 : n * fakultaet(n-1) ;
}
```

```
Int fak1(ulong n) {
    Int res = 1;
    while (n>1) { res = res * n--; }
    return res;
}
```

```
Int fak2(ulong n) {
    Int res = 1;
    while (n>1) { res *= n--; }
```

```
    return res;
}
```

```
Int fak3(Int n) {
    Int res = 1;
    while (n>1) { res *= n--; }
    return res;
}
```

```
Int ggt(Int a, Int b) {
    a = abs(a), b = abs(b);
    return b==0 ? a : ggt(b,a%b);
}
```

```
template <class S, class T>
S timing( S f(T), T n, char *s) {
    S x; int t;
    timer(0); x = f(n); t = timer(1);
    printf("Zeit (%s) : %d msec\n",s,t);
    return x;
}
```

```
int main() {
    long unsigned int n;
    int i, t=0;
    Int a[4];
    Int nn;
```

```
    while(1) {
        cout << " : "; cin >> n;
        if (n < 0) break;

        a[0] = timing(factorial,n,"factorial");
        a[1] = timing(fak1,    n,"fak1    ");
        a[2] = timing(fak2,    n,"fak2    ");
        nn    = n;
        a[3] = timing(fak3,    nn,"fak3    ");
```

```
        if ( ! (a[0] == a[1] && a[1] == a[2]) && a[2] == a[3] ) {
            printf("ERROR");
        }
        //cout << a[0] << "\n" ;
    }
}
```