世界の GMP 大全

— GMP の理論と実装（って書いておくとかっこいい）—

ykm11　著

世界のトリトリ祭り
2099 年 11 月 6 日　ver 1.4

■免責

本書は情報の提供のみを目的としています。
本書の内容を実行・適用・運用したことで何が起きようとも、それは実行・適用・運用した人自身の責任であり、著者や関係者はいかなる責任も負いません。

■商標

本書に登場するシステム名や製品名は、関係各社の商標または登録商標です。
また本書では、™、®、© などのマークは省略しています。

まえがき

本書を手に取っていただき、ありがとうございます。著者の ykm11 です。ゆうけむと読みます。ネットではトリと呼ばれることが多いです。

本書は、GMP(The GNU Multiple Precision Arithmetic Library)[*1]の実装を解説した本です。GMP を直訳すると GNU の多倍長精度の算術ライブラリ、つまり算術用の便利なライブラリです。便利なだけではなく、高速に動作するように実装されています。めちゃめちゃ速いです。速いと嬉しいですからね。

GMP を使用したソフトウェアの例としては、筆者がサイボウズ・ラボユースで開発していた楕円曲線暗号[*2]があります。

本書の目的

GMP や他の数値計算用のライブラリを使った数値計算を扱う書籍は存在する[*3]のですが、単にライブラリを用いて計算するだけに留まっていて、ライブラリがどのように実装されているかと深いところまで突っ込んだ書籍はまだありません。そこで、本書は GMP の実装を理解することを目的に執筆されました。GMP がどのように実装されているかを理解することで、多倍長精度の演算を自分で実装する際の助けになります。また、GMP のおかしな挙動に遭遇したときにどうデバッグすればいいのか、どの変数・メンバを見ればいいのかは GMP の知識で解決できます。そういうわけで、GMP ユーザが実装を理解しておくというのは大事なんですね。

GMP は巨大なプロジェクトですので、すべての実装を理解するには膨大な時間が必要になります。そのため、本書では筆者が適当に抜粋した関数や機能を解説します。ただ、基本的には、筆者が GMP を使っているときに「ここの実装どうなってるんだろうか」と気になったものを抜粋しています。本当に適当に選んだわけではないと弁明しておきます。

本書の対象読者

本書では次のような人を対象としています。

- 多倍長精度の整数, 有理数, 浮動小数点数を C/C++ で扱いたい人

[*1] https://gmplib.org/

[*2] https://github.com/ykm11/lab-youth/tree/master/ellipticCurve

[*3] 多倍長精度数値計算 GNU MP, MPFR, QD によるプログラミング https://www.morikita.co.jp/books/mid/085491

- GMP の実装に興味がある人（相当な変人かも）

本書は基本的に GMP の実装解説を取り扱いますが、はじめに使用例を示します。とりあえず使ってみたい！ という人は使用例を参考にしてみてください（使うだけならネットに情報がありそうな気もしますが・・）。

前提とする知識

本書を読むにあたり、次のような知識が必要となります。

- C/C++ 言語の基礎知識
- アセンブリの基礎知識

GMP のほとんどが C 言語で実装されています。C++ はそれほど出てきませんので、C 言語の知識があれば読み進められると思います。C++ に関しては class という概念と演算子オーバーロードさえ知っていれば問題ありません。また、アセンブリの基礎知識と書いていますが、高度な内容は扱いません（高度な内容ってなんだ）。そもそも筆者がそこまでアセンブリに詳しくないので、読めないよ！ って人でも大丈夫です。ガンガンいきましょう。

本書の配布形式

電子版のみで、無料公開としました（そもそもこの本の完成の目処が立っていないので印刷とかできません）。どうしてもぼくに投げ銭したいという方は Amazon の欲しい物リストから何か買ってくれると嬉しいです。

- Amazon wishlist: https://www.amazon.jp/hz/wishlist/ls/23CZA18QR1CQ0

問い合わせ先

本書に関する質問やお問い合わせは、筆者の Twitter までお願いします。

- URL: `https://twitter.com/_ykm11_`

謝辞

感謝するぜ、読者と出会えたこれまでの全てに。

目次

第 1 章

導入

本章では、GMP の導入と使用例を示します。GMP を使って多倍長整数の計算がしたい！ という人は本章を参考にしてください。また、本章で想定する実行環境は、Linux もしくは macOS となります。筆者の環境は Ubuntu18.04 です。途中でアセンブリを示すことがあり、objdump を使用するので、追実験を行いたい方は筆者と同じ Ubuntu18.04 の利用をおすすめします。例示するプログラムは、特にことわりがない限りは C++ です。

GMP のダウンロードとビルド

2021/01/24 時点での最新版である v6.2.1 をダウンロードします。configure の引数には、C++ 用のビルドを有効にする--enable-cxx フラグを与えます。ビルドオプションはこちら[*1]を参考にしてください。

macOS の場合、Homebrew を使って GMP をビルドすることも可能なのですが、Homebrew からではなくソースコードからビルドすることをおすすめします。

▼GMP のダウンロードとビルド

```
$ wget https://gmplib.org/download/gmp/gmp-6.2.1.tar.xz
$ tar -xvf gmp-6.2.1.tar.xz
$ cd gmp-6.2.1
$ ./configure --enable-cxx
$ make && make install
```

GMP の使用例

GMP が使用できる環境が整ったので、さっそく使ってみたいと思います。
多倍長整数のクラスである mpz_class の使用例を 2 つ示します。（有理数は mpq_class、浮動小数点数は mpf_class です。とりあえず本章では mpz_class のみということで。）コンパイルオプションに-lgmpxx -lgmp を渡し忘れないように注意しましょう。

[*1] https://gmplib.org/

▼GMP sample 1

```
#include <iostream>
#include <gmpxx.h>

int main() {
    mpz_class x = 1;

    for (size_t i = 1; i <= 50; i++) {
        x *= i;
    }
    std::cout << x << std::endl;
}
```

▼GMP を使ったプログラムのビルド 1

```
$ g++ source.cpp -lgmpxx -lgmp && ./a.out
30414093201713378043612608166064768844377641568960512000000000000
```

50 の階乗を計算してみました。64 ビットを超える値でも、きちんと計算できていることがわかります。プログラマはオーバーフローの可能性を考慮しなくてもいいわけです。また、四則演算も、プリミティブ型と同様に +-*/が使えます。便利ですね。

▼GMP sample 2

```
#include <iostream>
#include <gmpxx.h>

int main() {
    mpz_class e("65537", 10);
    mpz_class p("ecfc4ddf98ac14100230284ddf6f3a109995e74294070399857
76681a702eedd25683888f090a6f87778aeed537170ef2901644a560ae76273fccc5
b45aa9f97", 16);
    mpz_class q("edf5598b5a427b9c64ecd007e336e21eb3a93788b55c3f0cd13
7e2bbae554d35721b1fe6db65e9f0a23c3b963702ecb4fcab58882a0cdc161d1af57
1cf14a553", 16);
    mpz_class n = p*q;
    mpz_class c, m = 3;

    mpz_powm(c.get_mpz_t(), m.get_mpz_t(), e.get_mpz_t(), n.get_mpz_
t());
    std::cout << c << std::endl;
}
```

▼GMP を使ったプログラムのビルド 2

```
$ g++ source.cpp -lgmpxx -lgmp && ./a.out
111773376294672907599702415031428295845690629305468357913917896182470
180168633000957853136057180492094305963677700909479933552856831448350
749048864334518068590271012170336721192486365516045622783995514946620
933188699304169977202699403671712165132706561340874238875631162811970
151157595146921428448177985432568336 1
```

なんだかいきなりゴツくなりましたね。見た目はイカついですが、難しいことはしていません。mpz_class はコンストラクタに文字列と基数を受け取ることができます。e は 10 進数で 65537 を、p と q は、長いので省略しますが 16 進数で値を受け取っています。

mpz_powm はべき剰余の関数です。mpz_class ではなく mpz_t を引数に取るので、get_mpz_t() メソッドを呼び出しています。mpz_class 用のべき剰余関数は実装されていないため、mpz_t 用の関数を呼び出しているわけですね。

もうお気付きかもしれませんが、mpz_class というのは mpz_t の wrapper です。mpz_t に演算子を定義（実装）することで、プログラマにとって†いい感じに†書きやすくなっています。mpz_class は、厳密には"mpz_t の"wrapper ではないのですが（2 秒で嘘を吐く）、このへんは後々お話しましょう。

本章のまとめ

本章では、GMP の導入と mpz_class を使ったサンプルプログラムを示しました。使うだけなら意外と簡単ですよね（ですよね？）。次章からは、GMP の実装を見ていきます。C/C++ のプログラムを読んだり、自分でプログラムを書いてコンパイルしたオブジェクトファイルのアセンブリを読んだりします。乞うご期待！

第 2 章

mpz - 多倍長整数

本章では、多倍長整数の mpz_t と mpz_class を扱います。これからはプログラムを読んでいきます。

2.1 mpz_t

2.1.1 mpz_t を使う前に

mpz_class は mpz_t の wrapper じゃないの？

前の章で、「mpz_class は厳密には mpz_t の wrapper ではない」といいました。まずはこの件を解決しましょうか。
というわけで、gmp.h の中から mpz_t に関する記述を探します。

▼gmp.h

```
typedef struct
{
  int _mp_alloc;        /* Number of *limbs* allocated and pointed
                           to by the _mp_d field.  */
  int _mp_size;         /* abs(_mp_size) is the number of limbs the
                           last field points to.  If _mp_size is
                           negative this is a negative number.  */
  mp_limb_t *_mp_d;     /* Pointer to the limbs.  */
} __mpz_struct;

typedef __mpz_struct mpz_t[1];
```

見てわかるように、_mp_alloc, _mp_size, _mp_d というメンバを持つ構造体は、__mpz_struct という識別子で定義されていますね。その下で、typedef によって __mpz_struct のサイズ 1 の配列が mpz_t という型として定義されています。"厳密には"というのはこういうことです。表記として mpz_t を使っても問題ないので、

本書では__mpz_struct とは書かず、mpz_t と書くようにします（長いのは面倒とかそういう理由じゃないですよ）。
mpz_t は構造体のインスタンスではなくアドレスなので、メンバ変数にアクセスするときはドット演算子ではなくアロー演算子を使います。また、mpz_t を関数の引数として渡すときに、ポインタ渡しとなることに注意しましょう。

mpz_t のメンバ変数

mpz_t の各メンバ変数を説明します。

_mp_alloc
: _mpd_d がメモリ確保している領域のサイズ。

_mp_size
: _mp_d が実際に使用している領域のサイズ。_mp_d[alloc-1] から順に値を見ていって、最初に not 0 が見つかるインデックス。

_mp_d
: mp_limb_t 配列の先頭アドレス。

mp_limb_t は 32 ビットか 64 ビットの符号なし long int で、環境によって変わります。
_mp_size は負数を取ることもでき、負のときには_mp_d の値が負数であることを意味します。_mp_size の 1 ビットを使って、isNeg のようなフラグを実現しています。mpz_t インスタンスが使用するメモリサイズ（= sizeof(mpz_t)）が 16 バイトちょうどなので、アライメントがいい感じになるわけです。
また、計算の結果、桁数が増えるような場合（乗算に多い）は_mp_alloc, _mp_size が適宜更新されます。このとき realloc が走ります。

mpz_t が扱える最大値

多倍長整数といえども限界はあるはずです。1 つの mpz_t インスタンスが保持できる値の最大値の概算を出しましょう。
まず、_mp_size が領域サイズを表していて、符号付き int なので最大値は $2^{31}-1$ です。それから、64 ビット環境を想定すると、mp_limb_t は領域 1 つにつき 64 ビットです。したがって、mpz_t が保持できる値の最大値は、$64*(2^{31}-1) \approx 2^{37}$ ビットの値、つまり $2^{2^{37}}$ くらいです。デカすぎて固定資産税がかかりそうですね。ここまで大きな値を扱うことはそうそうないので、最大値を気にする必要はないと思います。

2.1.2　mpz_t を使ってみよう

では mpz_t を使ってみましょう。以下にコードを示します。mpz_init 関数, mpz_init_set_str 関数によって、mpz_t インスタンスが初期化されています。

▼mpz_t の使用例

```
#include <iostream>
#include <gmpxx.h>

int main() {
    mpz_t x, y, z;

    mpz_init(z);
    mpz_init_set_str(x, "d34d0000", 16);
    mpz_init_set_str(y, "b33f0000", 16);

    std::cout << x << std::endl;
    std::cout << y << std::endl;

    mpz_mul(z, x, y);
    std::cout << z << std::endl;
}
```

▼リスト 2.1: 実行結果

```
$ g++ source.cpp -lgmpxx -lgmp -O3 && ./a.out
3545038848
3007250432
10660819607104782336
```

mpz_t インスタンスが使用しているメモリ領域は free しないと開放されません。なんだか当たり前な気がしますが、放っておくとメモリが枯渇するので注意しましょう。追加の領域が必要になったときは勝手に realloc してくれるのですが、メモリ開放は自分で行う必要があります。メモリの開放には mpz_clear 関数を使います。
mpz_t インスタンスを初期化しなかった場合にどうなるか、気になりますよね? 次項でやります。

2.1.3　mpz_t インスタンスを初期化しないと・・

先に答えを言うと、初期化しなかったらプロセスが落ちます。何回も実行していると、稀に落ちることなく無事に終了することもあるのですが、まあバグなので潰しておきましょう。

実際にプロセスが落ちる様子をお見せします。

▼mpz_t を初期化しないとプロセスが落ちる

```
int main() {
    mpz_t x, y, z;

    //mpz_init(z);
    mpz_init_set_str(x, "d34d0000", 16);
    mpz_init_set_str(y, "b33f0000", 16);

    std::cout << x << std::endl;
    std::cout << y << std::endl;

    mpz_mul(z, x, y);
    std::cout << z << std::endl;
}
```

▼リスト 2.2: 実行結果

```
$ g++ source.cpp -lgmpxx -lgmp -O3 && ./a.out
3545038848
3007250432
zsh: segmentation fault (core dumped)  ./a.out
```

セグフォしましたね。計画どおりです。mpz_t インスタンスを初期化する前と後とで各メンバ変数がどうなっているのかをみてみます。

▼mpz_t を dump してみる

```
#include <iostream>
#include <gmpxx.h>

void mpz_dump(const mpz_t r) {
    printf("_mp_alloc = %d\n", r->_mp_alloc);
    printf("_mp_size  = %d\n", r->_mp_size);
    printf("_mp_d     = %p\n\n", r->_mp_d);
}

int main() {
    mpz_t x;

    mpz_dump(x);
    mpz_init(x);
```

```
    mpz_dump(x);
}
```

各メンバ変数を表示する mpz_dump という関数を実装しました。これを実行してみると、初期化後は_mp_alloc, _mp_size に 0 が、_mp_d にはスタック領域のアドレスが入っていることがわかります。初期化前は.. なんかすごいことになっていますよね。_mp_d が NULL です。この状態で mpz_の四則演算関数に mpz_t インスタンスを渡すと、NULL にアクセスすることになるのでセグフォが出ます。

どうして mpz_t インスタンスのメンバ変数に変な値が入っているのか、スタックポインタを知っている読者の方ならわかると思います。mpz_t インスタンスを宣言しただけでは、スタックポインタが sizeof(mpz_t) 分引き算されるだけです。このときの各メンバ変数の値はスタック次第ということになります。

▼リスト 2.3: 実行結果

```
$ g++ source.cpp -lgmpxx -lgmp -O3 && ./a.out
_mp_alloc = -1511941312
_mp_size  = 32546
_mp_d     = (nil)

_mp_alloc = 0
_mp_size  = 0
_mp_d     = 0x7f22a5bf1288
```

図 2.1 に mpz_t のメモリ配置図を示します。mpz_t 自体はスタック領域に乗りますが、_mp_d はヒープ領域を指しています。

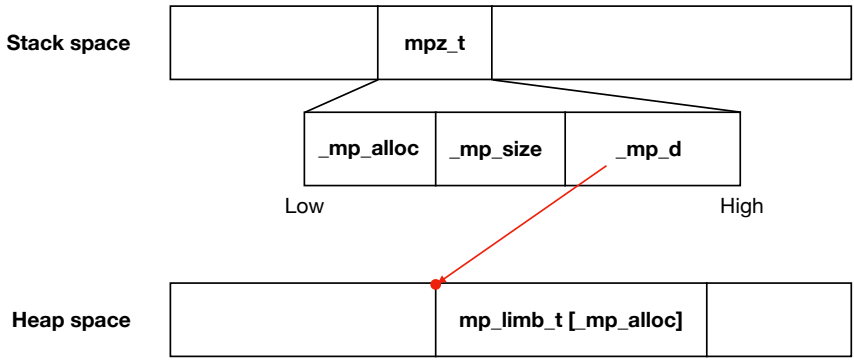
mpz_init の実装

mpz_init 関数の実装をみていきます。

▼mpz_init(mpz/init.c)

```
void
mpz_init (mpz_ptr x) __GMP_NOTHROW
{
  static const mp_limb_t dummy_limb=0xc1a0;
  ALLOC (x) = 0;
  PTR (x) = (mp_ptr) &dummy_limb;
  SIZ (x) = 0;
}
```

おっと、mpz_ptr なるものが出てきましたね。gmp.h で mpz_ptr を探すと、



▲図 2.1: mpz_t のアドレス

__mpz_struct*であることがわかりました。mpz_t もアドレスですので、mpz_init 関数に引数として渡せます。
ALLOC (x), PTR (x), SIZ (x) はマクロ定義です。動きについては説明しなくていいかもしれませんが、一応みておきます。gmp-impl.h で define されています。

▼mpz_ptr の typedef(gmp.h)

```
typedef __mpz_struct *mpz_ptr;
```

▼SIZ, PTR, ALLOC のマクロ定義 (gmp-impl.h)

```
#define SIZ(x) ((x)->_mp_size)
#define PTR(x) ((x)->_mp_d)
#define ALLOC(x) ((x)->_mp_alloc)
```

話を mpz_init 関数に戻します。_mp_alloc, _mp_size に 0 を代入して、_mp_d には static な定数のアドレスを代入しています。0xc1a0 という値の意味はわかりません。一旦初期化をしておけば、四則演算の関数に渡しても不正アクセスなどでプロセスが落ちることはありません。

mpz_init_set_str の実装

次に mpz_init_set_str 関数です。重要なのは最後の mpz_set_str 関数なのですが、ここを解説するには紙面と気力が足りないので、mpz_init_set_str の解説だけとさせてください。

▼ mpz_init_set_str (mpz/iset_str.c)

```
int
mpz_init_set_str (mpz_ptr x, const char *str, int base)
{
  static const mp_limb_t dummy_limb=0xc1a0;
  ALLOC (x) = 0;
  PTR (x) = (mp_ptr) &dummy_limb;

  /* if str has no digits mpz_set_str leaves x->_mp_size unset */
  SIZ (x) = 0;

  return mpz_set_str (x, str, base);
}
```

mpz_init_set_str は mpz_ptr と文字列、基数を引数に取ります。与えた文字列を基数で表現した値が必要とするメモリ確保したのち、確保したメモリの先頭アドレスを mpz_ptr->_mp_d に代入します。

mpz_init2 の実装

例に示した mpz_init, mpz_init_set_str 関数の他に、mpz_init2 というものがあります。mpz_init がただ初期値を与えるだけなのに対して、mpz_init2 ではあらかじめ決まったサイズのメモリ確保を行います。以下に mpz_init2 の実装を示します。

▼ mpz_init2 (mpz/init2.c)

```
void
mpz_init2 (mpz_ptr x, mp_bitcnt_t bits)
{
  mp_size_t  new_alloc;

  bits -= (bits != 0);          /* Round down, except if 0 */
  new_alloc = 1 + bits / GMP_NUMB_BITS;

  if (sizeof (unsigned long) > sizeof (int)) /* param vs _mp_size fi›
›eld */
    {
      if (UNLIKELY (new_alloc > INT_MAX))
        {
          fprintf (stderr, "gmp: overflow in mpz type\n");
          abort ();
```

```
        }
    }

  PTR(x) = __GMP_ALLOCATE_FUNC_LIMBS (new_alloc);
  ALLOC(x) = new_alloc;
  SIZ(x) = 0;
}
```

引数には mpz_ptr と、初期化時に確保する領域のサイズ（ビット数）を渡します。指定したビット数分のメモリ確保を過不足なく行うため、

$$\begin{aligned} new\_alloc := & \quad 1 + \frac{bits - 1}{GMP\_NUMB\_BITS} \\ = & \quad \frac{bits + (GMP\_NUMB\_BITS - 1)}{GMP\_NUMB\_BITS} \end{aligned}$$

という計算が行われています。bits=0 のときだけ特殊で、new_alloc=1 と定義されます。GMP_NUMB_BITS は次のように定義されています。

▼GMP_NUMB_BITS のマクロ定義 (gmp.h)

```
#if ! defined (__GMP_WITHIN_CONFIGURE)
#define __GMP_HAVE_HOST_CPU_FAMILY_power   0
#define __GMP_HAVE_HOST_CPU_FAMILY_powerpc 0
#define GMP_LIMB_BITS                      64
#define GMP_NAIL_BITS                      0
#endif
#define GMP_NUMB_BITS     (GMP_LIMB_BITS - GMP_NAIL_BITS)
```

__GMP_ALLOCATE_FUNC_LIMBS のマクロ定義は gmp-impl.h にあります。

▼__GMP_ALLOCATE_FUNC_LIMBS のマクロ定義 (gmp-impl.h)

```
__GMP_DECLSPEC extern void * (*__gmp_allocate_func) (size_t);

__GMP_DECLSPEC void *__gmp_default_allocate (size_t);

#define __GMP_ALLOCATE_FUNC_TYPE(n,type) \
  ((type *) (*__gmp_allocate_func) ((n) * sizeof (type)))
#define __GMP_ALLOCATE_FUNC_LIMBS(n)   __GMP_ALLOCATE_FUNC_TYPE (n, ›
›mp_limb_t)
```

__gmp_allocate_func がメモリ確保を行う関数で、ここを書き換えることにより、メモリ確保に好きな関数を利用できます。

以下に示すコードを使って、mpz_init2 関数で mpz_t を初期化したときの各メンバ変数の値を確認してみます。

▼mpz_init2 を使った初期化

```
#include <iostream>
#include <gmp.h>

void mpz_dump(const mpz_t r) {
    printf("_mp_alloc = %d\n", r->_mp_alloc);
    printf("_mp_size  = %d\n", r->_mp_size);
    printf("_mp_d     = %p\n", r->_mp_d);
}


int main() {
    mpz_t x;
    mpz_init2(x, 256);
    mpz_dump(x);
}
```

▼リスト 2.4: 実行結果

```
$ g++ source.cpp -lgmpxx -lgmp -O3 && ./a.out
_mp_alloc = 4
_mp_size  = 0
_mp_d     = 0x55af0395ce70
```

_mp_size は 0 ですが、_mp_alloc には 4 が代入されており、_mp_d_にはヒープ領域のアドレスが代入されていますね。

ついでに、__gmp_default_allocate で使われるメモリ確保の関数を調査します。上のプログラムを使って、gdb-peda[*1]でディスアセンブルします。

▼リスト 2.5: gdb で__gmp__default_allocate をディスアセンブル

```
gdb-peda$ disassemble __gmp_default_allocate
Dump of assembler code for function __gmp_default_allocate:
   0x00007ffff7b66970 <+0>:     push   rbx
   0x00007ffff7b66971 <+1>:     mov    rbx,rdi
   0x00007ffff7b66974 <+4>:     call   0x7ffff7b65cb0 <malloc@plt>
   0x00007ffff7b66979 <+9>:     test   rax,rax
```

[*1] https://github.com/longld/peda

```
   0x00007ffff7b6697c <+12>:    je     0x7ffff7b66980 <__gmp_default›
›_allocate+16>
   0x00007ffff7b6697e <+14>:    pop    rbx
   0x00007ffff7b6697f <+15>:    ret
   0x00007ffff7b66980 <+16>:    mov    rax,QWORD PTR [rip+0x26b659] ›
›       # 0x7ffff7dd1fe0
   0x00007ffff7b66987 <+23>:    mov    rcx,rbx
   0x00007ffff7b6698a <+26>:    mov    esi,0x1
   0x00007ffff7b6698f <+31>:    lea    rdx,[rip+0x540fa]        # 0x›
›7ffff7bbaa90
   0x00007ffff7b66996 <+38>:    mov    rdi,QWORD PTR [rax]
   0x00007ffff7b66999 <+41>:    xor    eax,eax
   0x00007ffff7b6699b <+43>:    call   0x7ffff7b66360 <__fprintf_chk›
›@plt>
   0x00007ffff7b669a0 <+48>:    call   0x7ffff7b651c0 <abort@plt>
End of assembler dump.
```

malloc が使われていました。macOS の標準 malloc は、Linux の malloc と比べるとかなり遅いので、Microsoft 社の mimalloc を使うといいでしょう。

mpz_clear の実装

mpz_t の初期化の話をしたので、ついでにメモリ解放にも触れておきます。
ではソースコード。

▼ mpz_clear(mpz/clear.c)

```
void
mpz_clear (mpz_ptr x)
{
  if (ALLOC (x))
    __GMP_FREE_FUNC_LIMBS (PTR (x), ALLOC(x));
}
```

ほとんどマクロですね。gmp-impl.h から__GMP_FREE_FUNC_LIMBS のマクロ定義を探します。

▼ (gmp-impl.h)

```
#define __GMP_FREE_FUNC_TYPE(p,n,type) (*__gmp_free_func) (p, (n) * ›
›sizeof (type))
#define __GMP_FREE_FUNC_LIMBS(p,n)     __GMP_FREE_FUNC_TYPE (p, n, m›
›p_limb_t)
```

メモリ解放の関数__gmp_free_func に x->_mp_d と (x->_mp_alloc * sizeof(mp_limb_t)) を渡して開放してもらっています。
メモリ確保、開放の関数はプログラマが自由に変更することができます。自作した malloc, free 関数を使ってみるのもいいかもしれませんね。

2.1.4　mpz 関数群

接頭辞に mpz_がつく関数をここでは mpz 関数と呼び、本節では mpz 関数の解説を行います。ソースコードをそのまま貼り付けると、プログラムのために数ページ割くことになるので、コメントやあまり重要でない箇所に関しては省略しています。プログラム全体を見たい方は本家のソースコードを参照してください。

mpz_add, mpz_sub

mpz_add, mpz_sub はそれぞれ加算、減算を行う関数です。まずソースコードを示してから、上から順に処理内容を解説していきます。表示する行数を削減するため変数宣言を省略していますが、型が明らかものばかりなので読解は難しくないと思います。

▼ (mpz/aors.h)

```
  1: #include "gmp-impl.h"

#ifdef OPERATION_add
#define FUNCTION     mpz_add
#define VARIATION
#endif
#ifdef OPERATION_sub
#define FUNCTION     mpz_sub
#define VARIATION    -
#endif

#ifndef FUNCTION
Error, need OPERATION_add or OPERATION_sub
#endif

void
FUNCTION (mpz_ptr w, mpz_srcptr u, mpz_srcptr v)
{
  usize = SIZ(u);
  vsize = VARIATION SIZ(v);
  abs_usize = ABS (usize);
```

```
abs_vsize = ABS (vsize);

if (abs_usize < abs_vsize)
  {
    /* Swap U and V. */
    MPZ_SRCPTR_SWAP (u, v);
    MP_SIZE_T_SWAP (usize, vsize);
    MP_SIZE_T_SWAP (abs_usize, abs_vsize);
  }

/* True: ABS_USIZE >= ABS_VSIZE.  */

/* If not space for w (and possible carry), increase space.  */
wsize = abs_usize + 1;
wp = MPZ_REALLOC (w, wsize);

/* These must be after realloc (u or v may be the same as w).  */
up = PTR(u);
vp = PTR(v);

if ((usize ^ vsize) < 0)
  {
    /* U and V have different sign.  Need to compare them to deter›
  ›mine
  which operand to subtract from which.  */

    /* This test is right since ABS_USIZE >= ABS_VSIZE.  */
    if (abs_usize != abs_vsize)
     {
     mpn_sub (wp, up, abs_usize, vp, abs_vsize);
  wsize = abs_usize;
   MPN_NORMALIZE (wp, wsize);
   if (usize < 0)
       wsize = -wsize;
}
    else if (mpn_cmp (up, vp, abs_usize) < 0)
 {
     mpn_sub_n (wp, vp, up, abs_usize);
   wsize = abs_usize;
   MPN_NORMALIZE (wp, wsize);
   if (usize >= 0)
```

```
      wsize = -wsize;
  }
      else
  {
       mpn_sub_n (wp, up, vp, abs_usize);
      wsize = abs_usize;
      MPN_NORMALIZE (wp, wsize);
      if (usize < 0)
          wsize = -wsize;
  }
    }
  else
    {
      /* U and V have same sign.  Add them.  */
      mp_limb_t cy_limb = mpn_add (wp, up, abs_usize, vp, abs_vsize)›
    ›;
      wp[abs_usize] = cy_limb;
      wsize = abs_usize + cy_limb;
      if (usize < 0)
 wsize = -wsize;
    }

  SIZ(w) = wsize;
}
```

mpz_add, mpz_sub の実装は aors.h に記述されていて、ソースコードを共有しています（処理がほぼ同じなので）。ifdef を駆使して mpz_add と mpz_sub をそれぞれ実装しています。

mpz_mul

mpz_mul は名前のとおり、乗算を行う関数です。先程と同様に、まずソースコードを示してから、上から順に処理内容を解説していきます。

▼ mpz/mul.c

```
 1: void
 2: mpz_mul (mpz_ptr w, mpz_srcptr u, mpz_srcptr v)
 3: {
 4:   usize = SIZ (u);
 5:   vsize = SIZ (v);
 6:   sign_product = usize ^ vsize;
 7:   usize = ABS (usize);
```

```
 8:   vsize = ABS (vsize);
 9:
10:   if (usize < vsize)
11:     {
12:       MPZ_SRCPTR_SWAP (u, v);
13:       MP_SIZE_T_SWAP (usize, vsize);
14:     }
15:
16:   if (vsize == 0)
17:     {
18:       SIZ (w) = 0;
19:       return;
20:     }
21:
22:   if (vsize == 1)
23:     {
24:       wp = MPZ_REALLOC (w, usize+1);
25:       cy_limb = mpn_mul_1 (wp, PTR (u), usize, PTR (v)[0]);
26:       wp[usize] = cy_limb;
27:       usize += (cy_limb != 0);
28:       SIZ (w) = (sign_product >= 0 ? usize : -usize);
29:       return;
30:     }
31:
32:   TMP_MARK;
33:   free_me = NULL;
34:   up = PTR (u);
35:   vp = PTR (v);
36:   wp = PTR (w);
37:
38:   wsize = usize + vsize;
39:   if (ALLOC (w) < wsize) {
40:     if (ALLOC (w) != 0) {
41:         if (wp == up || wp == vp) {
42:             free_me = wp;
43:             free_me_size = ALLOC (w);
44:         }
45:         else
46:             (*__gmp_free_func) (wp, (size_t) ALLOC (w) * GMP_LI›
   ›MB_BYTES);
47:     }
```

```
48:
49:     ALLOC (w) = wsize;
50:     wp = __GMP_ALLOCATE_FUNC_LIMBS (wsize);
51:     PTR (w) = wp;
52:   }
53:   else {
54:     if (wp == up) {
55:         up = TMP_ALLOC_LIMBS (usize);
56:         if (wp == vp)
57:             vp = up;
58:         MPN_COPY (up, wp, usize);
59:     }
60:     else if (wp == vp) {
61:         vp = TMP_ALLOC_LIMBS (vsize);
62:         MPN_COPY (vp, wp, vsize);
63:     }
64:   }
65:
66:   if (up == vp) {
67:       mpn_sqr (wp, up, usize);
68:       cy_limb = wp[wsize - 1];
69:   }
70:   else {
71:       cy_limb = mpn_mul (wp, up, usize, vp, vsize);
72:   }
73:
74:   wsize -= cy_limb == 0;
75:
76:   SIZ (w) = sign_product < 0 ? -wsize : wsize;
77:   if (free_me != NULL)
78:     (*__gmp_free_func) (free_me, free_me_size * GMP_LIMB_BYTES)›
   ›;
79:   TMP_FREE;
80: }
```

まず 4 行目から 8 行目にかけては、 u, v のサイズを取り出して、計算結果の正負を取得しています。

次に、 u, v のサイズを比較して、 $u > v$ となるように値を適宜スワップします。こうすることで、16 行目から 30 行目の処理において、 v のサイズだけを確認すればよくなります。v のサイズが 0 のとき（つまり $v == 0$ ）、計算結果は当然 0 なの

で、 w のサイズに 0 をセットして終了します。 v のサイズが 1 のときは、 u のサイズに 1 足した数だけメモリ確保を行い、mpn_mul_1 関数を実行して乗算を行います、mpn_mul_1 は本章の後半で説明しますが、処理内容を簡単に説明すると、多倍長整数 (mp_limb_t*) と符号なし整数 (mp_limb_t) の乗算です。一般に、 n 桁と m 桁の数の乗算結果は $n+m$ 桁です。したがって、 w のサイズは u のサイズに 1 を足した数になります。小さい方のサイズが 0 もしくは 1 のときは計算が比較的容易でした。ここからは、どちらのサイズも 2 以上のときの処理です。

50 行目から 64 行目では、計算結果を格納するメモリが十分に確保されているかを確認し、足りなければ領域を追加するために reaqlloc が走ります。この間の細かい処理について説明していきます。40 行目の if 文では、 w がメモリ確保を既に行っているかどうかを確認しています。既にメモリ確保をしている場合、現在確保している領域を開放してから、必要なサイズ ($usize + vsize$) のメモリ確保を行います。このとき、 w が u, v のどちらかと同じメモリを指していたらメモリ解放を保留します。 w と一緒に u, v の領域まで開放されしまったら後々の計算に影響が出るためです。mpz_mul(x, x, y) のような記述をしているときに該当します。

次に追加のメモリ確保が必要でないとき（53~64 行）です。 w が u , v と同じアドレスを指しているときは、新たにメモリ領域を確保して、値を退避させます。MPN_COPY は、アドレスとサイズを渡すと値をコピーしてくれる処理のマクロです。そこそこ長いので省略させてください。

66 行目のアドレス比較は、2 乗の計算を行う mpn_sqr 関数への切り替えるかどうかを判断しています。2 乗の計算は、普通に乗算を行うよりも 2 乗専用の関数/アルゴリズムを使ったほうが速いです。2 乗のアルゴリズムについては Intel さんがドキュメントを出しているので、興味があればそちら[*2]を参照してください。同じ桁数の乗算のコストを 1 とすると、2 乗のコストは 0.8 程度になるといわれています。アドレスが一致しない場合は mpn_mul 関数を実行して乗算を行います。mpn_mul 関数は本章の後半で登場します。

最後は、計算結果の符号をセットして、保留していたメモリ解放を行って終わりです。

2.2 mpz_class

多倍長整数を扱うのに便利な mpz_t でしたが、インスタンスは初期化が必要です

[*2] https://www.intel.com/content/dam/www/public/us/en/documents/white-papers/large-integer-squaring-ia-paper.pdf

し（初期化忘れたらプロセスが落ちる）煩わしいですよね。

2.2.1　コンストラクタ

1 章で mpz_class の使用例を示しました。初期化はコンストラクタがやってくれていましたね。mpz_class は mpz_t から煩わしさを取り除いた僕らの希望です。C++ に感謝しましょう。ありがとう！！

2.2.2　代入時の扱い

▼mpz_class move

```
#include <iostream>
#include <gmpxx.h>

int main() {
    mpz_class x(100);
    mpz_class y = x;

    printf("x->_mp_d = %p\n", (x.get_mpz_t())->_mp_d);
    printf("y->_mp_d = %p\n", (y.get_mpz_t())->_mp_d);
}
```

▼実行結果

```
$ g++ source.cpp -lgmpxx -lgmp -O3 && ./a.out
x->_mp_d = 0x55a3b9b4be70
y->_mp_d = 0x55a3b9b4be90
```

実行してみると、ちゃんと異なるアドレスが表示されました。 y の値が変わっても、 x の値が一緒に変わることはありません。
上のプログラムの実行ファイルをディスアセンブルし、コンストラクタ呼び出しに該当する部分を探します。

▼コンストラクタによる初期化

```
$ objdump -d -M intel
 (略)
 9e8:   e8 83 ff ff ff          call   970 <__gmpz_init_set_ui@plt>
 9ed:   48 89 de                mov    rsi,rbx
 9f0:   48 89 ef                mov    rdi,rbp
 9f3:   e8 48 ff ff ff          call   940 <__gmpz_init_set@plt>
 (略)
```

x が mpz_init_set_ui によって初期化され、次に y が mpz_init_set によって初期化されます。

2.3 mpn 関数群

これまでの節で mpz_t, mpz_class と解説してきました。mpz_t を wrap して使いやすくしたものが mpz_class でしたね。mpn 関数というのは、関数の接頭辞に mpn_がついたものをいいます。これまで見てきた mpz の関数内でも mpn 関数が使われています。また、mpn 関数は各アーキテクチャで可能な限り高速に動くよう設計されています。ほんとかなぁと思いますが、Low-level Functions[*3]に"The mpn functions are designed to be as fast as possible, ..."と書いてあります。mpz 関数の中で mpn 関数が使われているので、mpn 関数を使いやすくしたものが mpz_t だと考えることもできます。本節では、最もプリミティブな mpn 関数について解説します。

2.3.1 mpn の概要

mpn 関数は mpz 関数と違ってメモリ確保をやってくれません。プログラマが自分でメモリ管理を行います。使用するサイズの最大値がわかっていれば mp_limb_t の配列（スタック領域）をうまく使うことができ、malloc/free を呼び出す必要がないので、処理速度が上がります。サイズが不定の場合は、malloc を使って必要な分のメモリを確保（ヒープ領域）するようになるでしょう。これなら mpz_t にメモリ管理を任せたほうがいいですね。

使用するサイズの最大値が決まっている場合というのは、たとえば有限体を実装するときです。有限体の標数 (素数) が n ビットとすると、四則演算の結果は最大でも $2n$ ビットにしかなりません。このような場合は mpn 関数を使った実装による高速化が望めるでしょう。筆者が開発していた楕円曲線暗号では、mpz_class と mpn 関数の両方で実装しています。

2.3.2 mpn 関数の実装

mpn_mul_1

[*3] https://gmplib.org/manual/Low_002dlevel-Functions

mpn_mul

2.4 おまけ

この節では小ネタを紹介します。

スタック領域を使用する mpz_t

mpz_t インスタンスの _mp_d はヒープ領域を指しています、malloc を使ってメモリ確保するのですから当然ですよね。実は、_mp_d はスタック領域の配列を指すようにもできます。といっても、筆者は値を表示したいときくらいしか使いません。mpz_t インスタンスの各メンバ変数に値をセットする関数 set_mpz_t を以下のコードに示します。

▼mp_limb_t*を mpz_t にセット

```
#include <iostream>
#include <gmpxx.h>

void mpz_dump(const mpz_t r) {
    printf("_mp_alloc = %d\n", r->_mp_alloc);
    printf("_mp_size  = %d\n", r->_mp_size);
    printf("_mp_d     = %p\n", r->_mp_d);
}

void set_mpz_t(mpz_t r, const mp_limb_t *x, size_t n) {
    r->_mp_alloc = n;
    while(n > 0 && x[n-1] == 0) {
        --n;
    }
    r->_mp_size = n;
    r->_mp_d = (mp_limb_t *)x;
}

int main() {
    mpz_t t;
    mp_limb_t x[4] = {1, 2, 3, 0};
    set_mpz_t(t, x, 4);

    mpz_dump(t);
}
```

set_mpz_t には mpz_t インスタンスと mp_limb_t*、サイズを渡します。

▼実行結果

```
$ g++ source.cpp -lgmpxx -lgmp && ./a.out
_mp_alloc = 4
_mp_size  = 3
_mp_d     = 0x7ffd682cbb40
```

実行してみると、正しく値がセットされていることがわかります。_mp_d はスタック領域のアドレスを指していますね。各メンバ変数への値のセットを自分でやっているので、mpz_init を使って初期化する必要がありません。

注意すべき点は、_mp_d がスタック領域を指しているときに realloc/free を発生させないことです。douoble free というエラー文と共にプロセスが終了します。そもそも malloc で確保していないメモリ（スタック領域）を free するのが良くないですよね。

mpz_cmp のバグ

ver6.2.0 まで、mpz_t の比較関数 mpz_cmp にバグがありました。Release note には、"A possible overflow of type int is avoided for mpz_cmp on huge operands." とあります。オーバーロードの可能性があったようです。実際のコードをみてみましょう。

▼ver6.2.0 以前の mpz_cmp

```
int
mpz_cmp (mpz_srcptr u, mpz_srcptr v) __GMP_NOTHROW
{
  mp_size_t  usize, vsize, dsize, asize;
  mp_srcptr  up, vp;
  int        cmp;

  usize = SIZ(u);
  vsize = SIZ(v);
  dsize = usize - vsize; // 筆者コメント：ここでオーバーフローする
  if (dsize != 0)
    return dsize;

  asize = ABS (usize);
  up = PTR(u);
  vp = PTR(v);
  MPN_CMP (cmp, up, vp, asize);
```

```
  return (usize >= 0 ? cmp : -cmp);
}
```

mpz_cmp の機能は、2 つの mpz_t インスタンス u, v を取って、u=v なら 0 を、u>v なら正の数を、u<v なら負の数を返します。まず、u,v のサイズの差を計算しています。サイズ（ビット幅）の異なる値の比較が容易だからですね。

$$mpz\_cmp(u,v) = \begin{cases} 0 & (u = v) \\ Positive\ Num & (u > v) \\ Negative\ Num & (u < v) \end{cases}$$

具体例で説明します。usize が 10, vsize が 8 だとすると、明らかに u が大きいです。mpz_cmp は $10 - 8 = 2$ を返すことになり、正の値なので正しい動作となります。片方が負の場合で同様に確かめられます。

では、次の場合はどうでしょうか。usize を $2^{31} - 1$ とします。_mp_size は符号付きの int ですので、これは最大値です。ここで、vsize が負の数のときは $usize - vsize = 2^{31} - 1 + |vsize|$ でオーバーフローしてしまい、mpz_cmp は負の数を返します。想定外の動作となっています。_mp_size が $2^{31} - 1$ 程度の値を使うことはあまりないのですが、バグはバグです。次のように修正されました。

▼ ver6.2.1 の mpz_cmp

```
int
mpz_cmp (mpz_srcptr u, mpz_srcptr v) __GMP_NOTHROW
{
  mp_size_t  usize, vsize, asize;
  mp_srcptr  up, vp;
  int        cmp;

  usize = SIZ(u);
  vsize = SIZ(v);
  /* Cannot use usize - vsize, may overflow an "int" */
  if (usize != vsize)
    return (usize > vsize) ? 1 : -1;

  asize = ABS (usize);
  up = PTR(u);
  vp = PTR(v);
  MPN_CMP (cmp, up, vp, asize);
  return (usize >= 0 ? cmp : -cmp);
}
```

引き算をするとオーバーフローの可能性があるので、サイズの比較を行うようにしています。MPN_CMP は gmp-impl.h の中で__GMPN_CMP のマクロとして定義されていて、__GMPN_CMP の中身は gmp.h に記述されています。

$$mpz\_cmp(u,v) = \begin{cases} 0 & (u = v) \\ 1 & (u > v) \\ -1 & (u < v) \end{cases}$$

mpn_zero vs memset

mp_limb_t*をゼロ埋めするとき、mpn_zero を使う方法と memset を使う方法の 2 つがあります。for 文はひとまず忘れてください。どちらを使うべきかというのを、実行時間の面で決めることにします。速いほうが嬉しいので、実験で実行時間を測定してみます。実験に使用したコード群はここ[*4]に置いてあります。

mpn_zero と memset でそれぞれ、(1) 固定長でゼロ埋め、(2) 可変長でゼロ埋め、の 4 パターン用意しました。結果としては、固定長の memset が最も速く、次いで可変長の memset です。mpn_zero は固定長、可変長にかかわらず同じくらいです。

固定長の memset が速いのは、memset を呼び出さずに SIMD 命令（128 ビットレジスタなど）で 0 埋めができるからです。可変長の場合は memset を呼び出さざるをえないので、関数コールのコストが上乗せされます。SIMD 命令が使用できないマシンでは測定結果が変わるはずなので、古い PC などを所持している方は実験してみると面白いと思います。。

memset に関してはこのくらいにして、mpn_zero について見ていきましょう。mpn_zero の実装を以下に示します。

▼ mpn_zero の実装 (mpn/generic/zero.c)

```
void
mpn_zero (mp_ptr rp, mp_size_t n)
{
  mp_size_t i;

  rp += n;
  for (i = -n; i != 0; i++)
    rp[i] = 0;
}
```

引数にポインタとサイズを受け、for 文で順にアクセスして 0 をセットしています。これはよくある方法だと思います。ただ、コンパイル時にサイズが不定なので、for

[*4] https://gist.github.com/ykm11/7d8b48d628c281ebe21004572f2c41e7

文のループが何回繰り返されるかは実行するまでわかりません。そのため、この関数はベクトル化がしにくくなっています。

GMP をビルドすると zero.o というオブジェクトファイルが生成されるので、objdump を使ってアセンブリを見てみます。

▼ zero.o のアセンブリ

```
0000000000000000 <__gmpn_zero>:
   0:   48 89 f0                mov    rax,rsi
   3:   48 8d 14 f5 00 00 00    lea    rdx,[rsi*8+0x0]
   a:   00
   b:   48 f7 d8                neg    rax
   e:   48 85 f6                test   rsi,rsi
  11:   74 1a                   je     2d <__gmpn_zero+0x2d>
  13:   48 01 fa                add    rdx,rdi
  16:   66 2e 0f 1f 84 00 00    nop    WORD PTR cs:[rax+rax*1+0x0]
  1d:   00 00 00
  20:   48 c7 04 c2 00 00 00    mov    QWORD PTR [rdx+rax*8],0x0
  27:   00
  28:   48 ff c0                inc    rax
  2b:   75 f3                   jne    20 <__gmpn_zero+0x20>
  2d:   c3                      ret
```

zero.c のコードをそのままアセンブルに落とし込んだ内容になっていると思います。

ちなみにサイズが固定長のときの memset はコンパイルすると次のようになります。実験のコードをコンパイルしたものです。

▼ 固定長 memset

```
0000000000000000 <_Z3fffPm>:
   0:   66 0f ef c0             pxor   xmm0,xmm0
   4:   0f 11 07                movups XMMWORD PTR [rdi],xmm0
   7:   0f 11 47 10             movups XMMWORD PTR [rdi+0x10],xmm0
   b:   c3                      ret
```

128 ビットのレジスタである xmm0 レジスタを使っています。SIMD 命令が使用できるという条件付きですが、ゼロ埋めには memset を使うのが良いといえます。

GMP on M1 MBP

ver6.2.1 で、Arm ベースのアーキテクチャである M1 チップのサポートが始まりました。しかし、実は問題を抱えています。arm64 向けに実装されたコードで x18 レジスタを使うのですが、M1 において、x18 レジスタはリザーブされています。基本的に触っちゃいけないわけですね。テストも問題なく通るらしく、実行中にプロ

セスが終了するようなことはまだ報告されていません（2020.02.14 現在）。どういうコードを書けばプロセスが落ちるのかを研究するのも面白そうですね。

第 3 章

mpq - 多倍長有理数

本章では、多倍長有理数の mpq_t と mpq_class を扱います。

3.1 mpq_t

3.1.1 mpq_t を使う前に

mpq_t の定義を gmp.h から探します。

▼ mpq_t の定義 (gmp.h)

```
typedef struct
{
  __mpz_struct _mp_num;
  __mpz_struct _mp_den;
} __mpq_struct;

typedef __mpq_struct mpq_t[1];
```

mpz_t と同様に、__mpq_struct という識別子で構造体が定義されて、そのあとに mpq_t が typedef されています。メンバ変数は__mpz_struct の _mp_num と _mp_den ですね。分子と分母です。ということは、__mpz_struct さえわかってしまえば mpq_t は難しいものではないですね。

3.1.2 mpq_t の初期化

mpq_set_str

mpq_t に値をセットする mpq_set_str 関数を解説します。実装は以下のとおりです。

▼ mpq_set_str の実装 (mpq/set_str.c)

```
int
mpq_set_str (mpq_ptr q, const char *str, int base)
{
  const char  *slash;
  char        *num;
  size_t      numlen;
  int         ret;

  slash = strchr (str, '/');
  if (slash == NULL)
    {
      SIZ(DEN(q)) = 1;
      MPZ_NEWALLOC (DEN(q), 1)[0] = 1;

      return mpz_set_str (mpq_numref(q), str, base);
    }

  numlen = slash - str;
  num = __GMP_ALLOCATE_FUNC_TYPE (numlen+1, char);
  memcpy (num, str, numlen);
  num[numlen] = '\0';
  ret = mpz_set_str (mpq_numref(q), num, base);
  __GMP_FREE_FUNC_TYPE (num, numlen+1, char);

  if (ret != 0)
    return ret;

  return mpz_set_str (mpq_denref(q), slash+1, base);
}
```

mpz_init_set_str と同じように、mpq_t インスタンスと文字列、基数を渡します。文字列の中にはスラッシュ (/) を含めることができます。たとえば、mpq_set_str(q, "1/3", 10) と書くと、q->_mp_num に 1 が、q->_mp_den に 3 がセットされます。

文字列にスラッシュが含まれていた場合、スラッシュの前と後に分割して、mpz_set_str 関数を使って _mp_num と _mp_den にそれぞれ値をセットします。スラッシュが含まれない場合は、_mp_den には 1 をセットし、mpz_set_str 関数を使って _mp_num に値をセットします。

mpq_init_set_str ではなく mpq_set_str であることに注意してください。mpq_t は初期化と同時に初期値を与える関数がありません。mpq_init で初期化

してから mpq_set_str で値をセットします。

3.1.3　mpq 関数群

mpq_mul

有理数の乗算関数 mpq_mul の解説です。先に有理数の乗算を復習しましょう。op1 が $\frac{a}{b}$, op2 が $\frac{c}{d}$ で表され、それぞれ既約分数であるとします。これらの積 $op1 \cdot op2 = \frac{ac}{bd}$ が既約分数でないときは、約分して既約分数にする必要があります。約分せずに放置すると、無駄なメモリを使用する可能性があるからです。

まずは愚直な方法です。$\frac{ac}{bd}$ を既約分数で表すには、分子分母を $GCD(ac, bd)$ で割ればいいですよね。例) $\frac{3}{8} \cdot \frac{4}{9} = \frac{12}{72} = \frac{12}{72} \cdot \frac{\frac{1}{12}}{\frac{1}{12}} = \frac{1}{6}$

この方法の問題点は、乗算で桁数が増えてから除算を行うことです。mpz_t が確保するメモリ領域は増える一方なので、桁数はなるべく小さく抑えたいというモチベーションがあります。mpq_mul では、先に約分してから乗算するように実装されています。

では mpq_mul 関数の実装をみていきます。長いので、重要な部分に絞って解説します。残りの部分はメモリ確保のためのサイズ計算が主です。

▼ mpq_mul の実装を部分的に抜粋 (mpq/mul.c)

```
  mpz_gcd (gcd1, NUM(op1), DEN(op2));
  mpz_gcd (gcd2, NUM(op2), DEN(op1));

  mpz_divexact_gcd (tmp1, NUM(op1), gcd1);
  mpz_divexact_gcd (tmp2, NUM(op2), gcd2);

  mpz_mul (NUM(prod), tmp1, tmp2);

  mpz_divexact_gcd (tmp1, DEN(op2), gcd1);
  mpz_divexact_gcd (tmp2, DEN(op1), gcd2);

  mpz_mul (DEN(prod), tmp1, tmp2);
```

やはりマクロが出てきます。gmp-impl.h, gmp.h から NUM, DEN, mpq_numref, mpq_denref の定義を探します。

▼ NUM, DEN のマクロ定義 (gmp-impl.h)

```
#define NUM(x) mpq_numref(x)
#define DEN(x) mpq_denref(x)
```

▼numref, denref のマクロ定義 (gmp.h)

```
#define mpq_numref(Q) (&((Q)->_mp_num))
#define mpq_denref(Q) (&((Q)->_mp_den))
```

計算のアルゴリズムの触れていきます。
まず、 $t := GCD(a, d)$, $s := GCD(b, c)$ を計算しておきます。
$tmp1 := \frac{a}{t}$, $tmp2 := \frac{c}{s}$ と除算してから、 $NUM(prod) := tmp1 \cdot tmp2$ 。
分母も同様に、 t, s を使って、 $tmp1 := \frac{d}{t}$, $tmp2 := \frac{b}{s}$ と除算してから。
$DEN(prod) := tmp1 \cdot tmp2$ 。
正しく計算が行われていることを確認します。

$$\frac{NUM(prod)}{DEN(prod)} = \frac{\frac{a}{t}\frac{c}{s}}{\frac{d}{t}\frac{b}{s}} = \frac{ac}{bd} \cdot \frac{\frac{1}{ts}}{\frac{1}{ts}} = \frac{ac}{bd}$$

OK。除算で桁数を落としてから乗算するので、メモリを無駄遣いせずに済みます。
　op1=op2 のとき、つまり 2 乗のときの処理もついでに解説します。mpq_mul の上部に記述されています。

▼2 乗への切り替え (mpq/mul.c)

```
  if (op1 == op2)
    {
      /* No need for any GCDs when squaring. */
      mpz_mul (mpq_numref (prod), mpq_numref (op1), mpq_numref (op1)›
›);
      mpz_mul (mpq_denref (prod), mpq_denref (op1), mpq_denref (op1)›
›);
      return;
    }
```

　単に分子分母をそれぞれ 2 乗しているだけですね。シンプルです。ここで注意ですが、op1==op2 の比較は、値を比較しているのではなく、ポインタを比較しています。値が同じでもポインタが違う場合は通常の乗算が行われます。

3.2　mpq_class

第 4 章

mpf - 多倍長浮動小数点数

本章では、多倍長浮動小数点数の mpf_t と mpf_class を扱います。

4.1 mpf_t

4.1.1 mpf_t を使う前に

以下に__mpf_struct 構造体の定義を示します。実際の gmp.h には各メンバ変数についてコメントが記載されていますが、見やすさ重視のため、長いコメントに関しては削除しています。

▼mpf_t の定義 (gmp.h)

```
typedef struct
{
  int _mp_prec;         /* コメント削除 /*
  int _mp_size;         /* コメント削除 /*
  mp_exp_t _mp_exp;     /* Exponent, in the base of `mp_limb_t'.  */
  mp_limb_t *_mp_d;     /* Pointer to the limbs.  */
} __mpf_struct;

typedef __mpf_struct mpf_t[1];
```

mpf_t のメンバ変数

mpf_t の各メンバ変数を説明します。

_mp_prec

浮動小数点の精度（何ビットまで保証するか）を定める値。

_mp_size

_mp_d が実際に使用している領域のサイズ。_mp_d[0] から順に値を見ていって、最初に 0 が見つかるインデックス。

_mp_exp

浮動小数点の指数部に相当する。

_mp_d

mp_limb_t 配列の先頭アドレス。浮動小数点の仮数部に相当する。

4.1.2　mpf_t を使ってみる

自然対数の底ネイピア数 e の計算

ネイピア数 e の定義は $\lim_{t\to\infty}(1+\frac{1}{t})^t$ ですが、 $\lim_{t\to\infty}$ の表現が大変なので指数関数 e^x をマクローリン展開して近似します。

$$\begin{aligned} e^1 = &\quad 1+1+\frac{1}{2!}+\frac{1}{3!}+\frac{1}{4!}+\cdots \\ = &\quad \sum_{k=0}^{\infty}\frac{1}{k!} \end{aligned}$$

もちろん無限回の足し算は不可能なので、どこかで計算を打ち切ります。float や double を使っている場合は、扱える値の範囲の制限があるため足し算の回数はそこまで多くなりません。一方、mpf_t はコンピュータが許す限りの精度で計算できるので、ほしい精度が得られるまで繰り返せばよいです。

▼ ネイピア数の計算

```
#include <iostream>
#include <gmpxx.h>

int main() {
    mpq_t k, t;
    mpq_init(k);
    mpq_init(t);
    mpq_set_ui(k, 2, 1);
    mpq_set_ui(t, 1, 1);

    mpf_t e;
    mpf_init2(e, 1500);

    for (size_t i = 2; i < 250; i++) {
        mpz_mul_ui(&(t->_mp_den), &(t->_mp_den), i);
        mpq_add(k, k, t);
    }

    mpf_set_q(e, k);
```

```
    std::cout << "2.718281828459045235360287471352662497757247093699›
›9595749669676277" << std::endl;
    gmp_printf("%.409Ff\n", e);
}
```

上のソースコードをよく見ると mpz, mpq, mpf が全部登場しています。これまでの知識を総動員しているようでなんだか楽しいですね。mpq 関数には int や uint などで除算する関数がないので、分母に除数をかけています。

▼実行結果

```
$ g++ exp.cpp -lgmpxx -lgmp -O3 && ./a.out
2.718281828459045235360287471352662497757247093699599574966967627724
2.718281828459045235360287471352662497757247093699599574966967627724›
›076630353547594571382178525166427427466391932003059921817413596629042›
›357290033429526059563073813232862794349076323382988075319525101901152›
›738341879307021540891499348841675092447614606680822648001684774118532›
›742345442437107539077744992069551702761838606261331384583000752044932›
›382656029760673711320070932870912744374704723069697720931014169283682›
›190
```

4.2　mpf_class

さいごに

機会があれば、楕円曲線暗号の実装の TIPS を紹介する内容で同人誌を書くかもしれません。

世界の GMP 大全

GMP の理論と実装（って書いておくとかっこいい）

2099 年 11 月 6 日　ver 1.4

著　者　ykm11

(powered by Re:VIEW Starter)