# ユーザにごみ集めを意識させないCライブラリの設計法

西田 直樹　酒井 正彦　坂部 俊樹

名古屋大学大学院 工学研究科

〒464-8603 名古屋市千種区不老町

nishida@sakabe.nuie.nagoya-u.ac.jp
sakai@nuie.nagoya-u.ac.jp
sakabe@nuie.nagoya-u.ac.jp

あらまし　リストなどの可変サイズの資源を用いるプログラムにおいて、もっとも厄介な問題の一つは「ごみ集め」である。C 言語などの通常の言語ではごみの回収をプログラマがプログラム中に指示しなければならず、指示の誤りは時として重大なエラーを起こす。

そこで本稿では、ユーザがごみ集めを意識せずに利用が可能なライブラリを作成するための設計法を提案する。本方法は、代入によりポインタ変数が更新されて参照されなくなるデータの回収をマクロ化する指針を与える。また、関数合成の際に生ずる中間データの回収をライブラリ内で処理する指針を与える。本設計法に基づいて作成したライブラリにより本手法を評価し、その有効性を示す。

キーワード　ごみ集め、C 言語、ポインタ

# Designing Unlimited Size Resource C-Libraries Freeing Users from GC Annoyance

Naoki Nishida　Masahiko Sakai　Toshiki Sakabe

Graduate School of Engineering, Nagoya University

Furo-cho, Chikusa-ku, Nagoya 464-8603, Japan

nishida@sakabe.nuie.nagoya-u.ac.jp
sakai@nuie.nagoya-u.ac.jp
sakabe@nuie.nagoya-u.ac.jp

Abstract　The garbage collection (GC) problem is one of the most troublesome problems in programming with unlimited size resources like lists. In usual imperative languages such as C, we need to describe codes for GC explicitly. A mistake in the GC description sometimes causes significant errors, even if it is very trifling.

This paper proposes a method for designing libraries having libraries with GC encapsulated. The main ideas are (1) providing macros for collecting those areas which become not referred from anywhere after updating pointer variables, and (2) inserting modules that collect garbage areas produced by function compositions. We evaluate our method by applying it to libraries for variable length integers.

key words　Garbage Collection, C Language, Pointer

# 1 はじめに

リストなどの可変サイズの資源を用いるプログラムにおいてもっとも厄介な問題の一つは、ごみ集め (garbage collection)、すなわち、不要になったデータ領域の回収である。C 言語などの通常の言語ではごみの回収をプログラマがプログラム中に指示しなければならず、指示の誤りは時として重大なエラーを起こす。また、ごみの回収もれによって容易にメモリを消費し尽くす。簡単なテストプログラムにおいては問題が起こらないことが多いが、実用プログラムの場合には、ポインタ変数を使用するだけでもこれらの問題が生じることがある。

具体的に C 言語による次のようなデータ構造をとる複素数を扱うプログラムについて考えよう。

```
typedef struct complex{
    int real;
    int img;
} CPLX;
```

メイン関数において次のように処理したとする。

```
1: CPLX *x, *y;
2: x = make( 1, 2 );            /* x : 1+i2 */
3: y = make( 3, 4 );            /* y : 3+i4 */
4: y = add( y, negate( x ) );   /* y : 2+i2 */
5: ...
```

ただし、make( $r$ , $i$ ) は実数部が $r$ 、虚数部が $i$ となる複素数を作りそのポインタを返す関数、add( $x$ , $y$ ) は $x$ と $y$ を加算した結果の複素数へのポインタを返す関数、negate( $x$ ) は $x$ の符号を反転した複素数へのポインタを返す関数とする。

この一連の処理において、5 行目以降で参照される可能性があるのは変数 $x$ が保持するデータ（ $1+i2$ ）と変数 $y$ が保持するデータ（ $2+i2$ ）であり、3 行目で $y$ が保持したデータ（ $3+i4$ ）については、それを指すポインタ変数がなくなるためごみとして残る。さらに、4 行目の関数 add の引数である negate( $x$ ) の戻り値のポインタが指定するデータ（ $-1-i2$ ）も、関数 add で利用された後は、どこからも参照されずにごみとなっている。

これらの処理においてゴミを回収するためには、例えば次のような記述が必要である。

```
1: CPLX *x, *y, *p, *q;
2: x = make( 1, 2 );          /* x :  1+i2 */
3: y = make( 3, 4 );          /* y :  3+i4 */
4: p = y;                     /* p :  3+i4 */
5: q = negate( x );           /* q : -1-i2 */
6: y = add( y, q );           /* y :  2+i2 */
7: free( p );
8: free( q );
9: ...
```

ここで、関数 free は引数の指すデータのメモリを解放する標準ライブラリ関数である。

データ（ $3+i4$ ）へのポインタは 4 行目で一時変数 $p$ に格納され、6 行目の add でデータが使用された後、7 行目で回収される。また、データ ( $-1-i2$ ) へのポインタは 5 行目で一時変数 $q$ に格納され、6 行目の add でデータが使用された後、8 行目で回収される。

このような記述によりごみは残らなくなるが、最初のプログラムと比較すると記述量が格段に増える。また、ユーザは絶えずごみ集めを意識していなければならず、ごみの回収を忘れていても気付かない恐れもある。さらに、完成したプログラムがごみを残していないかを確認することが困難である。

そこで本稿では、ライブラリを利用してプログラムを行なうユーザ（以下では単にユーザと呼ぶ）がごみ集めを意識せずに利用できるライブラリの設計法を提案する。ユーザが通常記述する必要があるごみ集めはポインタ変数を更新する際の旧データの処理とライブラリ関数の合成の際に使われる中間データの処理である。これら 2 種類のデータの処理をユーザが記述しなくても処理されるようにするためにライブラリに必要な機能を明らかにし、その設計法を述べる。また、提案した設計法に基づいて C 言語で桁数に制限のない整数を処理する可変長整数ライブラリ、ならびにそのライブラリを利用したプログラムを作成し、手動でごみ集めを行うように修正したものとプログラミングの手間（行数・デバッグ時間）・ごみ集めの状況・実行時間の観点から比較し、本設計法に対する評価を行う。

また本稿では、可変サイズの資源は複数箇所から参照されない、すなわち 1 つのデータを指すポインタを持つ変数は 1 つしかないという制限のもとに議論を進める

# 2 ライブラリの設計法

## 2.1 設計の方針

本節では、ライブラリのユーザからごみ集めの記述を解放するために必要となるライブラリ内での処理についてまとめる。

### 2.1.1 ポインタ変数への代入の際の旧データの処理

ポインタ変数への代入 $x := expr$ によって参照ができなくなるデータを回収するためには、基本的には次の操作を順に行えばよい。

1. 式 $expr$ の計算
2. 変数 $x$ が参照するデータ領域の回収
3. 代入の実行

したがって、この一連の操作をユーザが通常の代入と同等の手間で記述できるように、これを実現する関数（以下では代入関数と呼ぶ）があればよい。ただし、ポイン

タ変数を宣言した直後の代入の場合には、実際には割り当てられていない領域を 2. で開放しようとするため問題が生ずる。そこで本手法では、ポインタ変数を宣言する際には必ず NULL に初期化するという制約を設けることによって、これを回避する。これにより、3. は一時変数が NULL でない場合にのみデータ領域が回収できるようになる。

#### 2.1.2 ライブラリ関数の合成の際の中間データの処理

ごみ集めの観点から見ると、ライブラリ関数に与えられる引数は、2 種類に分類される。1 つは変数にポインタが格納されている参照可能なデータであり、もう 1 つは他の関数の戻り値であって、変数にポインタが格納されていない参照可能でないデータである。後者の参照可能でないデータは呼ばれたライブラリ中でごみ集めをしない限り、回収は不可能である。しかし、前者の参照可能なデータは他で使用される可能性があるため、ライブラリ内で回収してはならない。

ライブラリ関数に渡された引数が指す領域が参照可能かどうかが分かるようにするため、参照可能であるかを示す情報（以下では参照状態と呼ぶ）をデータ構造に追加し、代入処理やライブラリ関数でこの情報を適正に管理する必要がある。これにより、ライブラリを終了する際に適切なゴミの回収が可能になる。

これらの引数に関する処理によって、ライブラリ関数の中で他のライブラリ関数を呼び出す場合は次の問題が生じる。ライブラリ関数 A からライブラリ関数 B を呼ぶ場合を考えると、A の引数の内で参照可能でないものが B に引数として与えられた場合、その引数が指定するデータは B の終了時に解放されてしまう。したがって、B から戻ってきた後で、そのデータを参照したときにはすでに解放されてしまっている。これを解決するために、各ライブラリ関数の中では、引数の指定するデータが参照可能であるかどうかの情報は一時変数に保持しておき、その関数の中では参照可能なデータとして振舞うようにする。

さらに、2.1.1 で述べたポインタ変数の更新の際において、代入されるデータを参照可能に設定する処理が必要となる。

## 2.2 ライブラリの設計法

以下では、具体的にライブラリを作成する際に、ごみ集めの機能を組み込む方法を述べる。本稿で主に対象とする可変サイズのデータは、しばしばデータ全体の情報を持つ一つのセル（以下では先頭セルと呼ぶ）と実際のデータを持つ複数のセル（以下ではデータセルと呼ぶ）で構成される。このため、以下では双方向リストでリストの長さの情報を持つデータ構造を例として用いる。しかしながら、本手法は、複素数の例のように単純なデータ構造の場合にも適用でき、その際新たにセルを設ける必要はない。

#### 2.2.1 データ構造の設計

> 可変サイズの資源のデータ構造を設計する際に、そのデータ領域が参照可能であるとき真となるフラグ（ referred ）を先頭セルに追加する。

双方向リストの例では、次のように先頭セルの型 blist とデータセルの型 bcell を定義する。

```
#define TRUE  1           /* 真 */
#define FALSE 0           /* 偽 */

typedef struct BCELL{     /* セル */
    int data;             /* データ */
    struct BCELL *upper;  /* 上位のセル */
    struct BCELL *lower;  /* 下位のセル */
} bcell;

typedef struct BLIST{     /* 双方向リスト */
    int len;              /* 長さ */
    struct BCELL *top;    /* 最上位のセル */
    struct BCELL *bottom; /* 最下位のセル */
    int referred;         /* 参照状態のフラグ */
} blist;
```

なお、参照状態は先頭セルにのみ付けておけば十分である。データセルについては、ユーザから代入や関数の引数として直接参照されないため参照状態が不要であるばかりか、データセルの参照状態に対する処理が実行速度の低下を招く。

#### 2.2.2 メモリ処理関数

通常行うように、メモリを割り当ててデータを初期化する関数（メモリ割り当て関数）、ごみを回収する関数（ごみ集め関数）、ならびに、データを複製する関数（複製関数）を用意する。

> メモリ割り当て関数については先頭セルを割り当てる関数のみが必須であり、その関数名を「make_型名」とし、参照状態を真に設定する以外は通常と同様に記述する。

この関数は、2.2.3 の代入処理で使われる。双方向リストの場合には、先頭セルのメモリ割り当て関数は次のようになる。

```
blist *make_blist(){
    blist *x;
    x = (blist *) malloc( sizeof( blist ) );
```

```
    x->len = 0;
    x->top = NULL;
    x->bottom = NULL;
    x->referred = TRUE;
    return( x );
}
```

ごみ集め関数「free_型名」はマクロと関数の組合せとして以下のように記述する。

```
#define free_型名 ( x ) (free0_型名 ( &(x) ))

void free0_型名 ( 型名 **xp ){
    if( *xp != NULL ){
        clear_型名 ( *xp );
        free( *xp );
        *xp = NULL;
    }
}
```

「free_型名」は引数 $x$ が NULL でなければ $x$ が指す先頭セルとデータセルの領域を回収する。さらに、データを指していない変数には NULL を入れておくという設計方針にしたがって、 $x$ に NULL を代入する。この処理を & なしに自然に使えるようにするため、「free_型名」をマクロ化した。

データセルを回収する処理は関数「clear_型名」として用意する。

この関数は通常と同様に記述すればよく、例えば、双方向リストの場合には次のようになる。

```
void clear_blist( blist *x ){
    bcell *y;
    if( x != NULL ){
        while( x->len > 0 ){
            y = x->top;
            x->top = y->lower;
            if( x->top != NULL )
                x->top->upper = NULL;
            free( y );
            x->len --;
        }
        x->bottom = NULL;
    }
    return;
}
```

また、上のメモリ割り当て関数やごみ集め関数の中で、実際にメモリ領域の操作を行う関数として、便宜上システム関数 malloc と free と用いたが、メモリ領域を管理するライブラリ関数を作成し用いても全く構わない。

複製関数は「copy_型名」の名前とし、通常と同様に作成する。ただし戻り値の参照状態は真に設定する。

例えば、双方向リストの場合には次のようになる。ここで make_bcell は bcell 型のメンバ data を引数の値で初期化する bcell 型のメモリ割り当て関数である。

```
blist *copy_blist( blist *x ){
    blist *y;
    bcell *xp, *yp;
    if( x == NULL ){
        return( NULL );
    }else{
        y = make_blist();
        xp = x->top;
        if( xp != NULL ){
            y->top = make_bcell( xp->data );
            y->bottom = y->top;
            y->len ++;
            xp = xp->lower;
        }
        while( xp != NULL ){
            yp = make_bcell( xp->data );
            y->bottom->lower = yp;
            yp->upper = y->bottom;
            y->bottom = yp;
            y->len ++;
            xp = xp->lower;
        }
        return( y );
    }
}
```

### 2.2.3 代入関数

2.1.1 で述べた処理によって、ゴミを回収しながら変数 $x$ に 式 $expr$ の結果を代入するための代入関数を定義する。

代入関数「assign_型名 ( $x$ , $expr$ )」は、マクロとして以下のように記述される。

```
#define assign_型名 ( x, expr ) ( ¥
                assign0_型名 ( &(x), expr ) ¥
)
```

ここで「assign0_型名」は実際に代入を行う関数であり、以下のように記述する。

```
void assign0_型名 ( 型名 **xp, 型名 *y ){
    if( *xp != NULL ){
        free_型名 ( *xp );
    }
    if( y == NULL ){
        *xp = NULL;
        return;
    }
    if( y->referred ){
        *xp = copy_型名 ( y );
    }else {
        *xp = y;
        (*xp)->referred = TRUE;
    }
    return;
}
```

代入されるポインタ変数のポインタを実際に代入を行なう関数「assign0_型名」に引き渡すことにより、関数により代入処理が行える。ただし、ポインタ変数のポインタ

の引き渡しをユーザに意識させないためにマクロを使用している。また、代入を関数内で行うことにより一時変数を用いることなくごみ集めが実現できている。なぜなら、マクロで定義した代入関数「assign_型名 ( $x$ , $expr$ )」の変数 $x$ が式 $expr$ 中で用いられていても、実際に代入を行う関数「assign0_型名」が呼び出された時点で式 $expr$ が評価されるため、変数 $x$ が保持していた旧データを代入関数内でいつでも回収してよいからである。

このほか、 $y$ が参照可能なデータの場合には、1 つのデータを指す変数は 1 つしかないという方針に基づきデータの複製が必要である。しかしながら、 $y$ が参照可能でないデータの場合には、データの複製は必要でなくデータを参照可能にするだけで十分である。

#### 2.2.4 ライブラリ関数の作成

2.2.1 から 2.2.3 で作成したユーティリティを用いて、ライブラリ関数を作成する。

ごみ集めの対象としているデータ型は、ポインタ形式でのみライブラリの引数や戻り値の型に出現してよい。そこで、ごみ集めの対象としているデータ型を blist とし、作成するライブラリ関数 f が以下の形式であるとして話を進める。

```
blist *f( blist *x0, blist *x1, int y )
{

    ライブラリ本体

    return( res );
}
```

1. blist のポインタ型の引数の数の要素を持つ整数型配列 ref を定義する。

   この例の場合には、以下のようになる。

   ```
   int ref[2];
   ```

2. blist のポインタ型の各々の引数 xi について、以下の文をライブラリ本体より前に記述する。

   ```
   ref[i] = xi->referred;
   xi->referred = TRUE;
   ```

3. blist のポインタ型の各々の引数 xi について、以下の文をライブラリ本体の後に記述する。

   ```
   xi->referred = ref[i];
   if( !(xi->referred) )
       free_blist( xi );
   ```

4. 戻り値が対象とするデータ型のポインタの場合には、戻り値のデータの参照状態を偽に設定する。

   この例の場合には return 文の直前に以下の文を記述する。

   ```
   res->referred = FALSE;
   ```

5. ライブラリ内で用いた対象とするデータ型のポインタの局所変数のうち、return で用いないものは、自動的には回収できないためデータを明示的に回収する。

   例えば、

   ```
   blist *z;
   ```

   で定義された局所変数 $z$ について、 return 文より前に

   ```
   free_blist( z );
   ```

   によってデータの回収が必要である。

これらの内で、1. から 3. により 2.1.2 で述べたゴミ集めの処理とライブラリ内でのライブラリ関数の呼び出しに関する問題を回避している。したがって、他のライブラリを利用していない場合には、参照状態の退避を記述する必要はない。

#### 2.2.5 ユーザ関数の作成法の指示

ユーザがライブラリを利用して独自の関数を作成する際には、2.2.4 のライブラリ関数の作成の際の形式に従って作成するように指示する。これにより、ユーザ作成関数におけるごみの回収もれを防ぐことができる。

## 3 設計法の評価

次のように本設計法に対する評価を行なった。

1. 設計法に基づく可変長整数ライブラリの作成
2. 任意に指定された桁数の $e$ の値を求めるプログラムの作成
3. 手動でごみ集めを行う可変長整数ライブラリとその応用プログラムの作成
4. これら 2 種類のプログラムに対する作成時の手間と動作時のごみ集めの状況・実行速度についての比較

手動でごみ集めを行なうライブラリおよび応用プログラムは、最初に作成したライブラリから本設計法に基づ

いている部分を修正することによって作成した。具体的には、一時変数を導入して関数合成を回避し、代入関数は使用せずに、代入の箇所ごとに旧データの解放とポインタの更新を行うように記述した。

## 3.1 可変長整数ライブラリの作成

提案した設計法に基づいて、可変長整数の比較・四則演算を行なうライブラリを作成した。なお、可変長整数のデータ構造・演算関数のアルゴリズム・内部関数内の処理方法は、Knuth の文献 [2] を参考にした。

実装に関しては、データ構造についてのみ簡単に述べる。

### 3.1.1 データ構造

データ構造は、可変長整数を 1000 進数として表現し、各桁をデータセルとする長さ付き双方向リストを用いた。双方向リストにした理由は、比較演算・四則演算によっては最上位の桁から走査していく場合と最下位の桁から走査していく場合があるので効率をよくするために必要だからである。

## 3.2 応用プログラムの作成

次に、可変長整数ライブラリを用いて指定された桁数の $e$ の値を求めるプログラムを作成した。

図 1 に示すのはそのプログラムの一部であり、e( $x$ , $keta$ ) は $keta$ が指定する桁数の $e^x$ の値の可変長整数を求め、そのポインタを返す関数である。このプログラムでは、ポインタ変数の宣言の際に初期化を行うこと、引数の参照するデータの参照状態を一時変数に退避すること、代入・演算を関数で書くこと、作成した関数の最後で不要なメモリを回収すること以外は、int 型整数で作成した場合（図 2）と内容は変わらず、図 3 の手動でごみ集めを行なう場合のプログラムと比較して、非常に記述性、読解性に優れている。図 1 と図 3 の行数はそれほど違わないが、図 3 の手動でごみ集めを行なうプログラムでは、関数 e を使用する際にもごみ集めの処理を記述する必要があるので、全体としてはプログラムの行数は本設計法に基づくプログラムより多くなる。

```
vint *e( vint *x, int keta )
{
    vint *E_i = NULL, *res_prev = NULL;
    vint *res = NULL, *i = NULL;
    int ref[1];

    ref[0] = x->referred;   /* 参照状態の退避 */
    x->referred = TRUE;

    assign_vint( E_i, one );
    for( ; keta > 0 ; keta -- )
        assign_vint( E_i, vmul( E_i, ten ) );

    assign_vint( res_prev, zero );
    assign_vint( res, E_i );

    for( assign_vint( i, one ) ;
            ! veq( E_i, zero ) ;
                assign_vint(i,vadd(i,one)) ){
        assign_vint( res_prev, res );
        assign_vint( E_i, vdiv(vmul(x,E_i),i) );
        assign_vint( res, vadd(res_prev,E_i) );
    }

    free_vint( E_i );    /* 局所データの回収 */
    free_vint( res_prev );
    free_vint( i );

    x->referred = ref[0];

    if( x->referred == FALSE ){
        free_vint( x );
    }      /* 引数の参照するデータの処理 */

    res->referred = FALSE;
              /* 戻り値の参照状態を偽に設定 */
    return( res );
}
```

図 1: 本設計法に基づく $e$ の値を求める関数プログラム

## 3.3 プログラミングにおける手間

ライブラリ作成時の手間を比較する。手動でごみ集めを行うライブラリは単純に代入・関数の合成の箇所を変更すればよいだけの作業ではあるが、手動ですべてのごみ集めを記述するのは大変困難であり、ライブラリの修正だけで、約 10 時間を要した。さらに、デバッグが困難であり、プログラムの行数も、設計法に基づいた場合に追加される分を考慮しても明らかに長くなった。もともと完成していたライブラリを修正するだけでこれだけの手間がかかったことから、本設計法の有効性がわかる。また、デバッグについては、本設計法による方が明らかに楽であった。

次に、設計法に基づいたライブラリを利用して応用プログラムを作成する際の手間を比較する。3.2 でも述べたように $e$ の計算プログラムは、int 型整数で作成した場合とほぼ同等の行数で記述できる。これは、明らかに設計法に基づくライブラリを使用したことによりプログラミングが簡単に行えると言ってもよい。さらに、ユーザは 2.2.4 で述べた関数の作成形式に従う箇所以外ではごみ集めを意識する必要がない。また、これらの操作は関数の作成方法として形式化し、指示されていることなので、実際にはユーザはまったくごみ集めを意識しなくてもすむ可能性もある。逆に、手動でごみ集めを行うライブラ

```
int e( int x, int keta )
{
    int E_i, res_prev, res, i;

    E_i = 1;
    for( ; keta > 0 ; keta -- )
        E=i *= 10;

    res_prev = 0;
    res = E_i;

    for( i = 1 ; E_i != 0 ; i ++ ){
        res_prev = res;
        E_i = ( x * E_i ) / i ;
        res = res_prev + E_i;
    }

    return( res );
}
```

図 2: int 型整数での $e$ の値を求める関数プログラム

リを使用した $e$ の計算プログラムでは明らかに記述が困難であった。この場合は、ユーザが絶えずごみ集めを意識していなければならず、さらにプログラムが正常に作動するまでかなりのデバッグ時間を要した。また、動いた時点でごみ集めをチェックすると、かなりの回収もれがあった。実際、ライブラリの修正と並行しながらこのプログラムを作成したわけだが、それでもごみが残るプログラムになってしまった。一般には、ユーザがごみの回収もれに気付かない可能性も考えられる。

以上のプログラミングの際の手間の比較の結果を表 1 にまとめる。

表 1: プログラミングにおける手間

| | 本設計法 | 手動 |
|---|---|---|
| デバッグ時間 | 少 | 多 |
| ライブラリ行数 | 約 2,800 行 | 約 3,000 行 |
| $e$ の計算プログラム行数 | 約 90 行 | 約 120 行 |

### 3.4 ごみ集めの状況

ごみ集めの状況については、使用中のメモリのアドレスを表示することにより調べた。これにより、どちらのプログラムもきちんとごみ集めを行なっていることがわかった。しかし、手動でごみ集めを行なうライブラリ・プログラムの方はかなりのデバッグを行なった後にようやくごみをすべて回収するようにできた。

```
vint *e( vint *x, int keta )
{
    vint *E_i, *res_prev, res, *i;
    vint *p;

    E_i = make_vint();
    E_i = copy_vint( one );
    for( ; keta > 0 ; keta -- ){
        p = E_i;
        E_i = vmul( E_i, ten ) );
        free_vint( p );
    }
    res_prev = make_vint();
    res_prev = copy_vint( zero );
    res = make_vint();
    res = copy_vint( E_i );

    i = make_vint();
    for( i = copy_vint( one ) ;
            ! veq( E_i, zero ) ;
                ({ p = i;
                   i = vadd( i, one );
                   free_vint( p ); }) ){
        free_vint( res_prev );
        res_prev = copy( res );
        p = vmul( x, E_i );
        free_vint( E_i );
        E_i = vdiv( p, i );
        free_vint( p );
        free_vint( res );
        res = vadd( res_prev, E_i );
    }

    free_vint( E_i );    /* 局所データの回収 */
    free_vint( res_prev );
    free_vint( i );

    return( res );
}
```

図 3: 手動でごみ集めを行なう $e$ の値を求める関数プログラム

### 3.5 実行速度

最後に、本設計法に基づいたライブラリと手動でごみ集めを行なうライブラリ、それぞれを利用した場合における $e$ の計算プログラムの実行速度を比較した結果を表 2 に示す。実行においては同一マシンにより同一条件下で計測した。また、 $e$ を求める計算アルゴリズムも同様とした。

この結果から考えて、本設計法に基づいて設計した方が、わずかながら実行時間が短くなることがわかる。これは一時変数に対する処理の時間が増加したためと思われる。

表 2: 実行速度の比較

| 桁数 | 本設計法 | 手動 |
|---|---|---|
| 100 | 0 秒 05 | 0 秒 06 |
| 1,000 | 2 秒 45 | 2 秒 67 |
| 10,000 | 2 分 56 秒 56 | 3 分 13 秒 22 |

# 4 まとめ

本稿では、可変サイズの資源を用いてプログラミングを行なう際のごみ集めの問題に対して、ユーザがごみ集めを意識せずに利用が可能なライブラリの設計法を提案した。さらに、これに基づいて実際に可変長整数のライブラリ・応用プログラムを作成することにより、提案した設計法の有効性を示した。本設計法ではライブラリの作成には、部分的にはごみ集めを意識してプログラミングをする必要はあるものの、次の利点を持つ。

- ユーザは意識せずにきちんとごみ集めを行なうことが可能。これはライブラリ作成時にも部分的にではあるが効果がある。
- デバッグ時間の減少
- プログラムの行数の減少
- プログラムの記述性・読解性の向上
- 手動によるごみ集めと同程度の実行速度

また、その他の C 言語におけるごみ集めの研究として、Boehm GC[3]、GC FAQ[4] といったものがある。今後の課題としてこれらの手法との比較を検討している。

本稿では、データが複数箇所から参照されないものとして話を進めてきたが、参照状態フラグを参照カウンタに置き換えて、代入関数とごみ集め関数で適切な処理を行うことによって容易に拡張可能である。

熱心にご討論頂いた坂部研究室の方々に感謝する。

# 参考文献

[1] B.W. カーニハン, D.M. リッチー著, 石田晴久訳: “プログラミング言語 C 第 2 版”, 共立出版株式会社

[2] Donald E. Knuth: “The Art of Computer Programming, Volume 2/Seminumerical Argorithms”, Addison-Wesley, Reading, Mass., 1969

[3] Boehm GC
URL: http://reality.sgi.com/boehm/gc.html

[4] GC FAQ
URL: http://www.iecc.com/gclist/GC-faq.html