3.4 クイックソート

配列と再帰の復習としてクイックソートを取り上げる。
クイックソートのアルゴリズムは以下のとおり。

1. ソートする範囲の中から適当な値 (基準値) を一つ選ぶ.
2. 配列内の要素を一つずつ調べて，基準値より小さなデータを配列の左側 (配列インデックスの小さい方)，大きなデータを配列の右側 (配列インデックスの大きな方) に集める. この操作を分割と呼ぶ.
3. 分割した対象に対して，さらにクイックソートのアルゴリズムを適用する.
4. 全ての分割が終ると，ソートが終了している.

クイックソートは高速なアルゴリズムとして知られている。

```
#include <stdio.h>

void swap(int a, int b, int x[])
{
    int tmp;

    tmp = x[a];
    x[a] = x[b];
    x[b] = tmp;
}

void qsort(int left, int right, int x[])
{
    int pivot, pivot_value, p, i;

    if ( left < right ) {
        pivot = (left + right)/2;  // 最初はド真ん中
        pivot_value = x[pivot];    // ピボットの設定
        x[pivot] = x[left];        // ピボットを選んだ場所に一番左の要素を代入
        p = left;                  // p に一番左のインデックスを保存

        for ( i = left + 1; i<= right; i++ ){
            if (x[i] < pivot_value ) { // x[i] がピボットより小さければ
                p++;                   // 値を入れるインデックスを計算し
                swap(p, i, x);         // x[p] と x[i] の値を交換
            }
        }
        x[left] = x[p];                // x[left] に x[p] を代入
        x[p] = pivot_value;            // ピボットの値を x[p] に代入
        // x[left] から x[p-1] は x[p] 未満，x[p+1] から x[right] は x[p] 以上となる
```

```
        qsort(left, p-1, x);
        qsort(p+1, right, x);
    }
}

int main(void)
{
    int x[]={3,1,2,6,5,4,7,9,8,0}, SIZE, i;

    SIZE = sizeof(x)/sizeof(x[0]);
    qsort(0, SIZE, x);
    for (i=0; i< SIZE; i++){
        printf("%d ", x[i]);
    }
    printf("\n");
    return 0;
}
```

3.5 フィボナッチ数列

フィボナッチ数列とは,

- n=0 のとき $f(0) = 0$
- n=1 のとき $f(1) = 1$
- その他の場合 $f(n) = f(n-1) + f(n-2)$

を満たす数列である。具体的には $0, 1, 1, 2, 3, 5, 8, 13, 21, 34, \ldots$ となる。フィボナッチ数列は, $f(n)/f(n-1)$ が黄金比約 1.618033　になることが知られている。このことをプログラムを作って確かめると次のようになる。

```
#include <stdio.h>
#include <math.h>

int main(void)
{
    int i=1;
    int fibonacci(int n);
    const double golden_ratio = 1.618033;
    double r;

    while(1) {
        r = (double)fibonacci(i)/(double)fibonacci(i-1) ;
        printf("Fibonacchi(%d)=%d(%f)\n",i,fibonacci(i),r);
        if ( fabs(r - golden_ratio) < 0.000001 ) {
            break;
```

```
        }
        i++;
    }

    return 0;
}

int fibonacci(int n)
{
    if ( n == 0 ) {
        return 0;
    } else if ( n == 1 ) {
        return 1;
    } else {
        return fibonacci(n-1) + fibonacci(n-2);
    }
}
```

4　ポインタ

4.1　アドレス

プログラムの実行中、オブジェクトや配列の値はメモリにおかれる。メモリにはアドレス (番地) がついている。アドレスの付け方はシステムにより異なるが、通常は 8 ビット (1 バイト) 毎にアドレスが付けられ、16 進数で表される。

オブジェクトのアドレスは、オブジェクトの前に**アドレス演算子**&をつけて表す。printf() 関数によるアドレスの表示は変換指定子%p を用いる。アドレスは 16 進で表示される。

例 4.1 オブジェクトのアドレス

```
/* address.c */
#include <stdio.h>

int main(void)
{
  int a;

  a = 1;

  printf("a の値は%d です。\n",a);
  printf("a のアドレスは%p です。\n",&a);    /* &はアドレス演算子
*/

  return 0;
}
```

コンパイルし実行すると、

```
~/comp3a$ cc address.c -o address -Wall
~/comp3a$ ./address
a の値は 1 です。
a のアドレスは 0xbffff40c です。
```

演習 4.1 address.c を変形し char 型のオブジェクト c1、c2、int 型のオブジェクト i1、i2、long long int 型のオブジェクト lli1、lli2、float 型のオブジェクト f1、f2、double 型のオブジェクト d1、d2、long double 型のオブジェクト ld1、ld2、を宣言しアドレスを調べてみよ。宣言する順序を変えるとどうなるか。

演習 4.1 より、同じ型のオブジェクトのアドレスは連続してとられること、オブジェクトの型により占有するメモリの大きさが異なることがわかる。

演習 4.2 演習 4.1 より、char 型、int 型、long long int 型、float 型、double 型、long double 型のオブジェクトはメモリ上何バイト占めていると考えられるか。

4.2　ポインタ

オブジェクトや関数のアドレスを格納するオブジェクトをポインタ型 (pointer type) という。ポインタの宣言は次の形式を用いる。

```
    型 *識別子;
```

ポインタ p がオブジェクト x のアドレスを保持している場合、「p は x を指している」という。

例 4.2 ポインタとポインタの指すオブジェクト

```
/* pointer.c */
#include <stdio.h>

int main(void)
{
  int a;
  int *pa; /* pa は int 型へのポインタ */

  a = 1;
  pa = &a; /* a のアドレスを pa に代入 */

  printf("a の値は%d です。\n",a);
  printf("a のアドレスは%p です。\n",&a);
  printf("pa の値は%p です。\n",pa); /* a のアドレスに一致 */
  printf("pa のアドレスは%p です。\n",&pa);

  return 0;
}
```

コンパイルし実行すると、

```
~/comp3a$ cc -Wall pointer.c -o pointer
~/comp3a$ ./pointer
a の値は 1 です。
a のアドレスは 0xbffff40c です。
pa の値は 0xbffff40c です。
pa のアドレスは 0xbffff408 です。
```

オブジェクト a とポインタ pa のアドレスと値の変化を示すと次のようになる。

```
6   int a;
```

で、0xbffff40c 番地に int 型オブジェクト a を割り当てる。

```
7   int *pa;
```

で、0xbffff408 番地に int 型へのポインタ pa を割り当てる。ポインタ pa の型は、”int へのポインタ”型であって、int 型ではない。

アドレス	オブジェクト	値
0xbffff40c	a	未定
0xbffff408	pa	未定

```
9   a = 1;
```

で、a に 1 を代入する。

アドレス	オブジェクト	値
0xbffff40c	a	1
0xbffff408	pa	未定

```
10   pa = &a;
```

で、ポインタ pa は int 型オブジェクト a を指している。pa に a のアドレス 0xbffff40c が代入される。

アドレス	オブジェクト	値
0xbffff40c	a	1
0xbffff408	pa	0xbffff40c

4.3 間接演算子

ポインタにオブジェクトのアドレスが格納されると、ポインタから元のオブジェクトの値を知ることができる。ポインタの指すオブジェクトの値を与える演算子*を間接演算子【間接参照演算子】(indirection operator) という。

例 4.3 pointer.c の pa から a の値を求める。

```
/* indirection.c */
#include <stdio.h>

int main(void)
{
  int a;
  int *pa;

  a = 1;
  pa = &a;

  printf("a の値は%d です。\n",*pa); /* '*' は間接演算子 */
  printf("a のアドレスは%p です。\n",pa);

  return 0;
}
```

ポインタを用いて、オブジェクトに値を代入することもできる。pointer.c の 9-10 行目

```
a = 1;
pa = &a;
```

を

```
pa = &a; /* pa に a のアドレスを代入 */
*pa = 1; /* *pa に 1 を代入 */
```

に置き換えても pointer.c と同じ結果になる。*pa はポインタ pa の指すオブジェクトの値を間接参照している。ただし、順序を変えて

```
*pa = 1; /* pa は何も指していないので誤り */
pa = &a;
```

とすると実行時に Bus error や Segmentation fault が生じることがある。

演習 4.3 上記のことを確かめよ。

ポインタ演算子、間接演算子

- オブジェクト x が〜型のとき、&x の結果は〜型へのポインタ (値は x のアドレス) になる。
- ポインタ p が〜型へのポインタのとき、*p の結果は〜型である。

例 4.4 ポインタを用いてオブジェクトの値を変更することができる。

```
/* pointer2.c */
#include <stdio.h>

int main(void)
{
  int a,b;
  int *pa; /* pa は int 型へのポインタ */

  a = 1;
  b = 2;
  pa = &a; /* a のアドレスを pa に代入 */

  printf("a の値は%d です。\n",a);
  printf("a のアドレスは%p です。\n",&a);
  printf("pa の値は%p です。\n",pa);
  printf("pa のアドレスは%p です。\n",&pa);

  pa = &b; /* b のアドレスを pa に代入 */

  printf("b の値は%d です。\n",b);
  printf("b のアドレスは%p です。\n",&b);
  printf("pa の値は%p に変更されました。\n",pa);
  printf("pa のアドレスは%p のままです。\n",&pa);

  return 0;
}
```

11 行目でポインタ pa は、a を指す。18 行目で b のアドレスを pa に代入しているので、a の値が b の値に変更される。

4.4 参照渡し

オブジェクトを引数として関数に渡すときは、オブジェクトの値が仮引数にコピーされるだけで、オブジェクト自体が関数に渡されるわけではないので呼び出した関数によりオブジェクトの値が書き換えられることはない。このような引数の渡し方を値渡し (pass by value) という。

```
/* pass-by-value.c */
#include <stdio.h>
int pbv(int n, int m);

int main(void)
{
  int a=1,b=2;

  printf("a=%d b=%d です。\n",a,b);
  a = pbv(a,b); /* pbv() の呼び出し */
  printf("a=%d b=%d です。\n",a,b);

  return 0;
}

int pbv(int n, int m)
{
  n += 10;
  m += 10;

  return n;
}
```

仮引数 m は関数 pbv() により変更されているが、実引数 b の値は変化しない。

一方、オブジェクト自体を関数に渡し関数の中でオブジェクトの値を変更するには参照渡し (pass by reference) という方法をとる。関数にオブジェクトのアドレスを渡すことにより実現できる。

```
/* swap.c */
#include <stdio.h>
void swap(int *p, int *q);

int main(void)
{
  int a,b;

  a = 1;
  b = 2;

  printf("a=%d b=%d です。\n",a,b);

  swap(&a; &b); /* swap() の呼び出し */

  printf("a=%d b=%d です。\n",a,b);

  return 0;
}

void swap(int *p, int *q)
{
  int tmp;

  tmp = *p;
  *p = *q;
  *q = tmp;
}
```

関数 swap() により、a と b の値が入れ替わっている。

演習 4.4 swap.c の a と b をキーボードから入力するように修正せよ。

演習 4.5 演習 4.4 を double 型のデータ用に修正せよ。

参照渡し

オブジェクトの値を関数の中で操作したいときは、

- 仮引数にオブジェクトのポインタ
- 実引数にオブジェクトのアドレス

を与える。関数がアドレスの内容を書き換えれば、オブジェクトの値も変わる。

関数で複数個のオブジェクトの値を操作したいときは、参照渡しを使うとよい。(1 個のオブジェクトの場合は返却値により変更できる。)

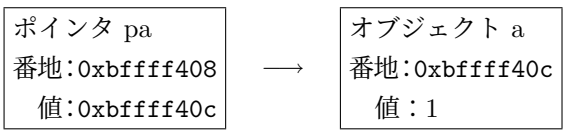
4.5　ポインタの算術演算

ポインタは

- 指しているオブジェクトのアドレス
- 指しているオブジェクトの型

を持つ。

pointer.c を図示すると



p をあるオブジェクトを指すポインタとしたとき、p++ により、その型の次のオブジェクトを格納またはアクセスするのにふさわしいアドレスが取得できる。char 型へのポインタなら 1 バイト、int 型へのポインタなら 4 バイト上位番地 (16 進数で表したとき値の大きい番地が上位番地、小さい番地を下位番地という。) に移動する。(システムにより移動するバイト数は異なる。) ++p、p+=1 も同様である。

ポインタに作用する算術演算子は、+、-、++、--の 4 つがある。ポインタは、アドレスという正整数の値を持つので整数のみの加算と減算が可能である。

演算子	ポインタ演算の例 (p、q はポインタ)	
+	p+1	p が指している要素の次の要素のアドレスを与える。
-	p-1	p が指している要素の前の要素のアドレスを与える。
	p-q	p と q の間の要素の数を与える。
++	p++	p が指している要素の次の要素のアドレスを与える。
--	p--	p が指している要素の前の要素のアドレスを与える。

ポインタに対する演算子の優先順位は、ポインタ演算子*、&は算術演算子より強く、演算子*、&、++、--は右から左に計算される。例えば、*p++は++が p の後に付いているため*(p++) と解釈されるので、ポインタをインクリメントする前のアドレスにある値を表す。(*p をインクリメントするには (*p)++とする。) また、*++p は*(++p) と解釈されるので、ポインタをインクリメントした後アドレスにある値を表す。

```
/* pointer-arithmetic.c */
#include <stdio.h>

int main(void)
{
  int a,b,*p;

  a = 1;
  b = 2;
  p = &b;

  printf("a のアドレス=%p b のアドレス=%p\n",&a,&b);
  printf("p の値=%p *p=%d\n",p,*p);
  printf("(p+1) の値=%p *(p+1)=%d\n",p+1,*(p+1));
  printf("*++p=%d\n",*++p);
  printf("p の値=%p *p=%d\n",p,*p);

  return 0;
}
```

出力結果は、

```
~/comp3a$ cc -Wall pointer-arithmetic.c -o pointer-arithmetic
~/comp3a$ ./pointer-arithmetic
a のアドレス=0xbffff408 b のアドレス=0xbffff404
p の値=0xbffff404 *p=2
(p+1) の値=0xbffff408 *(p+1)=1
*++p=1
p の値=0xbffff408 *p=1
```

4.6 配列とポインタ

添字指定を行わずに配列の名前を使うと、実際には、**配列の先頭アドレスを値に持つ定数** (ポインタ型の定数) を使っていることになる。

```
/* array-pointer.c */
#include <stdio.h>

int main(void)
{
  int a[3]={100,200,300};
  int *p;

  p = a;

  printf("p の値=%p a のアドレス=%p a の値 (16 進数)=%#x\n",
         p,a,(unsigned int) a);
  printf("*p=%d *a=%d a[0]=%d\n",*p,*a,a[0]);
  printf("*(p+1)=%d *(a+1)=%d a[1]=%d\n",*(p+1),*(a+1),a[1]);
  printf("*(p+2)=%d *(a+2)=%d a[2]=%d\n",*(p+2),*(a+2),a[2]);

  return 0;
}
```

実行結果は次のようになる。

```
~/comp3a$ cc -o array-pointer -Wall array-pointer.c
~/comp3a$ ./array-pointer
p の値=0xbffff400 a のアドレス=0xbffff400 a の値 (16 進数)=0xbffff400
*p=100 *a=100 a[0]=100
*(p+1)=200 *(a+1)=200 a[1]=200
*(p+2)=300 *(a+2)=300 a[2]=300
```

9 行目の p = a; は p に配列 a の先頭アドレスを代入している。したがって、p の値と a のアドレスは等しく (11 行目)、*p と*a と a[0] は等しくなる (12 行目)。

配列名 a は、先頭番地のアドレスであるがポインタではないので代入したり、インクリメントすることはできない。ポインタ型の定数なので、unsigned int 型にキャストすると、変換指定子%x、%X、%#x、%#X を用いて 16 進数で表示することができる。(11 行目) キャストをしないと、-Wall をつけてコンパイルした場合、警告「
warning: format '%#x' expects type 'unsigned int', but argument 4 has type 'int *'」がでる。

a+1 は a[0] の次の要素 a[1] のアドレスであるが、a++や++a は正しくない。

p+1 と a+1 は等しく*(p+1) と*(a+1) と a[1] は等しい (13 行目)。

ポインタ p が配列 a を指しているときは、ポインタ p に p[2] のように添字を当てることができる。[] を添字演算子 (subscript operator) という。p[2] はポインタ p が指している要素から 2 つ先の要素を表している。

演習 4.6 array-pointer.c の printf() 文の a[0],a[1],a[2] を p[0],p[1],p[2] に変更し実行してみよ。

ポインタ演算を使って配列を操作できる。

```
/* array-pointer2.c */
#include <stdio.h>

int main(void)
{
  int a[3]={100,200,300};
  int *p,i;

  p = a;

  for(i = 0;i < 3; i++){
    *p = (i + 1) * 10;
    printf("p=%p a+%d=%p\n",p,i,a+i);
    p++;
  }

  for(i = 0;i < 3; i++)
    printf("a[%d]=%d",i,a[i]);
  printf("\n");

  return 0;
}
```

p = a; で配列 a の先頭番地が p に代入される。12 行目の p は a[i] を指しているので、12 行目は a[i] = (i + 1) * 10; と同じである。

14 行目の p++; により、p は 4 番地後のオブジェクト a[i+1] を指す。

ポインタが配列を指しているとき、ポインタと配列名の関係は次のようになる。表の p と a は、下記の array-pointer3.c を例にとっている。

	ポインタ p		配列 a	
配列の先頭要素のアドレス	p	&p[0]	a	&a[0]
配列の先頭要素	*p	p[0]	*a	a[0]

次のプログラムで確かめることができる。

```
/* array-pointer3.c */
#include <stdio.h>

int main(void)
{
  int a[3]={100,200,300};
  int *p;

  p = a;

  printf("a[0] のアドレス:  p=%p &p[0]=%p a=%p &a[0]=%p\n",
      p, &p[0], a, &a[0]);
  printf("a[0] :  *p=%d p[0]=%d *a=%d a[0]=%d\n",
      *p, p[0], *a, a[0]);

return 0;
}
```

演習 4.7 array-pointer3.c を変更し、a[1] のアドレスと値を出力するようにせよ。(Hint. p+1 と a+1 は何を表しているか考えよ。)

4.7 引数に配列を持つ関数 — その 2

3.1.6 節 (p.31) では、配列 int a[6] を引数に持つ関数は、仮引数に int a[]、または int *a とし、実引数は配列名 a とすると述べた。参照渡し (4.4 節 p.49) で説明したように、仮引数の int *a は配列 a をさすポインタ、実引数の a は配列 a のアドレスである。

配列の要素の和を求める関数を考える。

```
/* array-sum.c */
#include <stdio.h>
#define N 10
double arraysum(double *a, int n);

int main(void)
{
  int i;
  double a[N];

  for(i = 0;i < N;i++){
    printf("%d 番目は",i);
    scanf("%lf",&a[i]);
  }

  printf("合計は%f です。\n", arraysum(a,N));

  return 0;
}

double arraysum(double *a, int n)
{
  int i;
  double sum = 0.0;

  for(i = 0;i < n;i++)
    sum += a[i];

  return sum;
}
```

1. 仮引数は `double a[]` とできる。
2. 関数呼び出しで `arraysum(a,3)` とすると、`a[0]+a[1]+a[2]` が返される。
3. 関数呼び出しで `arraysum(&a[2],4)` とすると、`a[2]+a[3]+a[4]+a[5]` が返される。

演習 4.8 上記のことを確かめよ。関数呼び出しで `arraysum(a+3,2)` とするとどうなるか。

演習 4.9 selection-sort.c をポインタを用いて書き換えよ。

演習 4.10 演習 3.15 の関数をポインタを用いて書き換えよ。