2.2　配列

(1)配列の考え方

同じデータ型の集まり(図 2-1 参照)を配列(Array)と呼び,次のように宣言します。

■宣言方法

データ型[] 配列変数名 = new データ型[要素数];

■データの代入:

配列変数名[インデックス] = データ;

[例]

```
Int[] A = new int[30];
A[0]=70; A[1]=77;
```

また，以下のように，要素数を宣言しないで，値をまとめて初期化することもできます。

[形式１]

データ型[] 配列変数名;
配列変数名 = new データ型[]{データ，データ，…，データ}

[例]

```
int[] A; A = new int[]{ 1, 2, 3, 4}
```

[形式２]

データ型[] 配列変数名 = {データ，データ，…，データ}

[例]

```
string[] A = {" Zero"," One"," Two"," Three"}
```

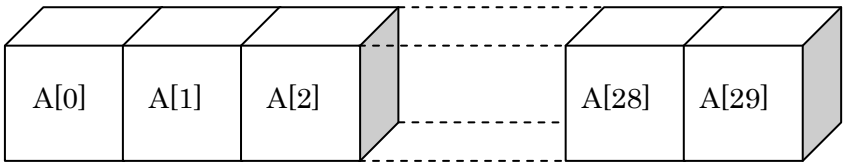


図 2-1　同じデータの集まり（配列）

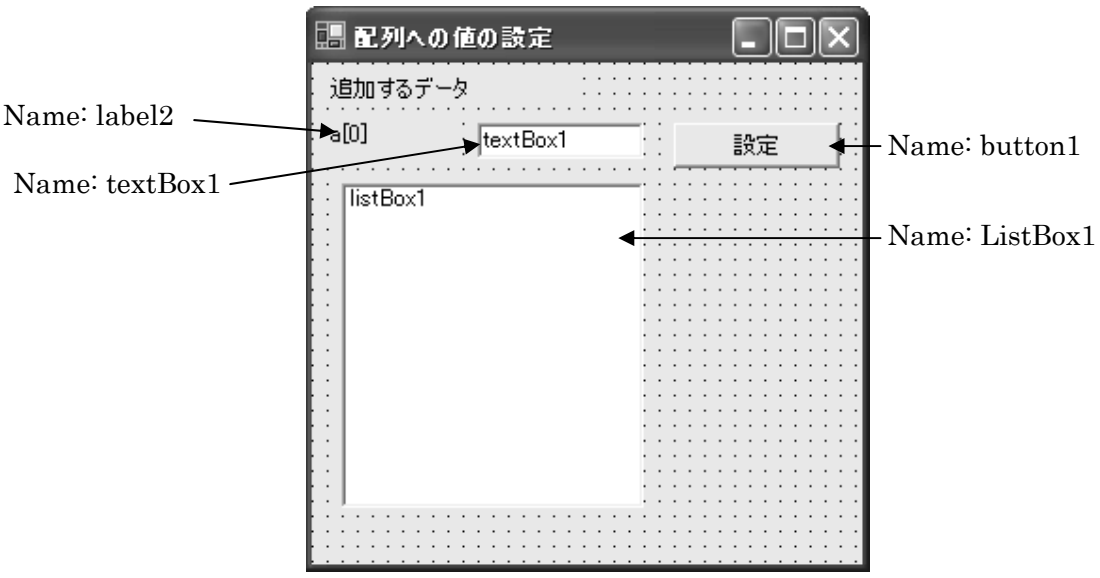
なお, 配列に初期値を設定して表示するプログラムを Program 2-2 に, 配列にデータを設定して表示する例を Program 2-3 に示します。

[Program 2-2] 配列の初期値設定例

Form にテキストボックス(Name：textBox1)を配置し, textBox1 の MultiLine プロパティを True にします。

```
private void Form1_Load(object sender, System.EventArgs e)
{ int i;
  int[] a       = new int[]{20,26,11,32,68};
  int na        = a.Length;
  textBox1.Text = "要素数 : " + na.ToString() + "¥r¥n";
  for (i = 0; i < na; i++)
      textBox1.Text += "¥r¥n a[" + i.ToString() + "] = "
                                 + a[i].ToString();
}
```

[Program 2-3] 配列へのデータ設定例



```
      //宣言分部
private int 配列数=0;
private int[] 配列 =new int[11];
   .
   .
   .
private void 配列の表示()
{   label2.Text="a[" + 配列数.ToString() + "]";
    listBox1.Items.Clear();
    for(int i=0; i<配列数; i++)
        listBox1.Items.Add("a[" + i.ToString()+ "]=" +配列[i]);
}
private void button1_Click(object sender, System.EventArgs e)
{   try
    {   if(配列数>10)MessageBox.Show("配列数が 10 を超えました");
        else
        {
            配列[配列数++]= int.Parse(textBox1.Text);
            配列の表示();
        }
     }
     catch( Exception myError)
     {
        MessageBox.Show(myError.Message);
     }
}
private void Form1_Load(object sender, System.EventArgs e)
{
    textBox1.Text="";
    listBox1.Items.Clear();
}
```

(2)最大値を求める

三値の最大値を求める方法の延長で，配列要素の最大値を求める問題を考えると，Program 2-4 のようになります。しかし，この方法では配列の要素数が増えるたびに行数が増えてしまいます。

そこで，値を入れ換える部分を次のように変えてみましょう。

```
i=1; if (MAX<a[i]) MAX = a[i];
i=2; if (MAX<a[i]) MAX = a[i];
i=3; if (MAX<a[i]) MAX = a[i];
i=4; if (MAX<a[i]) MAX = a[i];
i=5; if (MAX<a[i]) MAX = a[i];
```

すなわち，i =1～5 と変化させることで，if 文の部分が共通になります。したがって，次のように単純化することができます。

```
for(int i=1; i<6;i++) if (MAX<a[i]) MAX = a[i];
```

さらに，配列のサイズは Length プロパティを使って

```
配列名.Length
```

で取り出すことができますから，

```
for(int i=1; i<a.Length;i++) if (MAX<a[i]) MAX = a[i];
```

とすることで，要素数を気にしない表現ができます。改善した例を Program 2-5 に示します。

[Program 2-4] 最大値を求める単純な例

```
private int 最大値(int[] a)
{ int MAX = a[0];
  if (MAX<a[1]) MAX = a[1];
  if (MAX<a[2]) MAX = a[2];
  if (MAX<a[3]) MAX = a[3];
  if (MAX<a[4]) MAX = a[4];
  if (MAX<a[5]) MAX = a[5];
  return MAX;
}
private void button1_Click(object sender, System.EventArgs e)
{ int[] a = new int[]{50,20,33,55,44,25};
  int b   = 最大値(a);
  MessageBox.Show(b.ToString());
}
```

[Program 2-5] 最大値を求める(改善例)

```
private int 最大値(int[] a)
{ int i; int MAX = a[0];
  for(i = 1; i < a.Length; i++) if (MAX < a[i]) MAX = a[i];
  return MAX;
}
private void button1_Click(object sender, System.EventArgs e)
{ int[] a = new int[]{50,20,33,55,44,25};
  int b   = 最大値(a);
  MessageBox.Show(b.ToString());
}
```

(3)配列要素の逆転

配列要素の順序を逆転させるには，図 2-2 のように，最初と最後，2 番目と最後から 1 つ前，さらに 3 番目と最後から 2 つ前，・・・と交換していきます。

2 つの値を交換するには，通常，2 つの要素以外の作業的な変数を用意して，図 2-3 のような手順で行います。

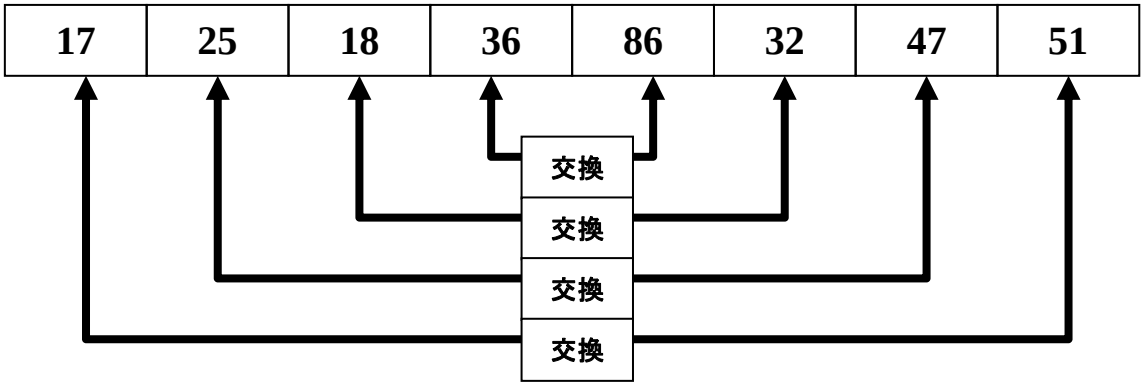


図 2-2　配列の逆転

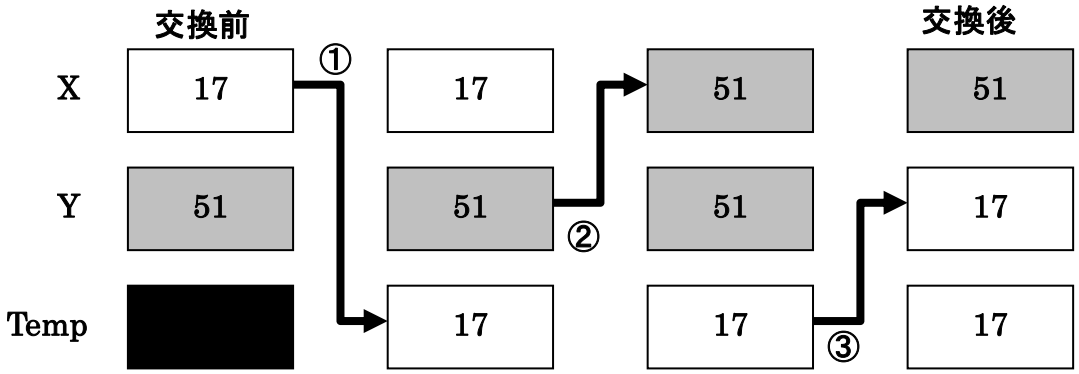


図 2-3　値の交換方法

値を交換するような手続きを考える際，C#では注意が必要です。旧来のCやC++では,標準的な引数の受け渡しは**参照(Reffer)渡し**ですが，C#の標準では**値(Value)渡し**です。

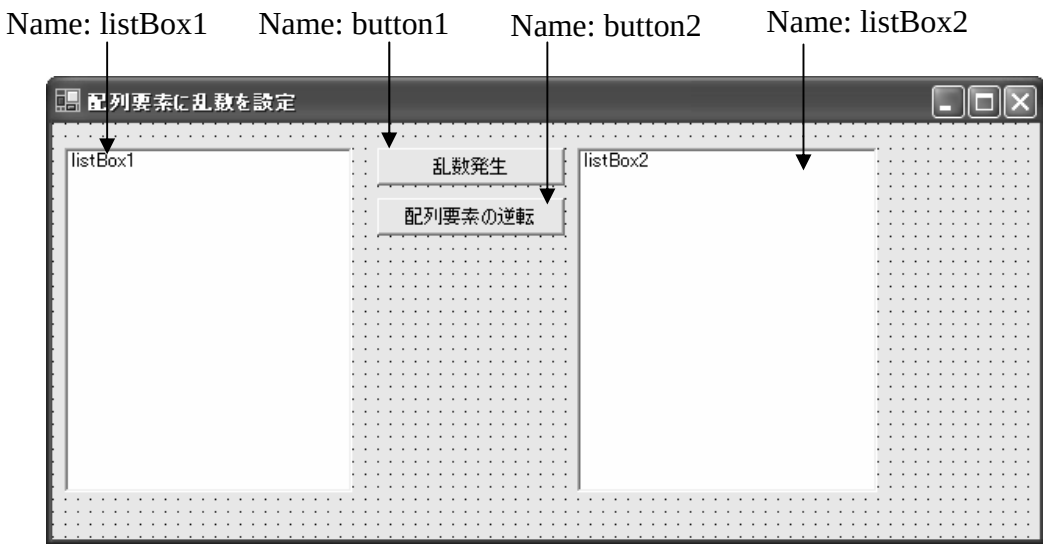
これは，呼び出された側の代入等で，呼び出し側の定数領域等が壊されることを避けるための処置です。したがって，特に何も指定しなければ，呼び出された関数側で値を設定しても，呼び出した側の値は変更されません。

呼び出した側の値を変更するような場合，呼び出す側，呼び出される側ともに積極的に参照渡しであることを指定します。

```
[呼び出す側]　　swap( ref a[i], ref a[n-i-1])
[呼び出される側]　private void swap( ref int X, ref int Y) { ・・・ }
```

なお，プログラム例では，データ入力の手間を省くため，逆転させる前のデータは乱数を発生させて設定しています(button1_Click)。

[Program 2-6] 配列の逆転



```
private int [] myArray = new int[10];
private void 配列の表示(ListBox A)
{
    A.Items.Clear();
    for(int i=0; i< myArray.Length; i++)
        A.Items.Add("a[" + i.ToString()+ "]=" +myArray[i]);
}
private void button1_Click(object sender, System.EventArgs e)
{
    Random myRandom = new Random();
    for (int i = 0; i < 10;i++) myArray[i] = myRandom.Next(0,100);
    配列の表示(listBox1);
}
private void swap( ref int X , ref int Y)
{
    int D = X; X = Y; Y = D;
}
private void 配列要素の逆転(ref int[] a)
{
    int n = a.Length;
    for (int i = 0; i < n / 2; i++) swap( ref a[i], ref a[n-i-1]);
}
private void button2_Click(object sender, System.EventArgs e)
{
    配列要素の逆転(ref myArray);
    配列の表示(listBox2);
}
```

(4)素数を求める

(a) 力まかせ法

素数の定義どおりに，その値より小さい値で割り切れないことを検査して，素数かどうかを判定する方法です。**ブルータス法**(Brutus Method)とも呼ばれます。

すなわち，2 よりも大きく着目する整数よりも小さい整数で除算を実行し，割り切れた整数があったら素数でないと判定し，次の整数値の判定に移ります。すべて割り切れなかったら素数とみなします。その過程を図 2-4 に示します。

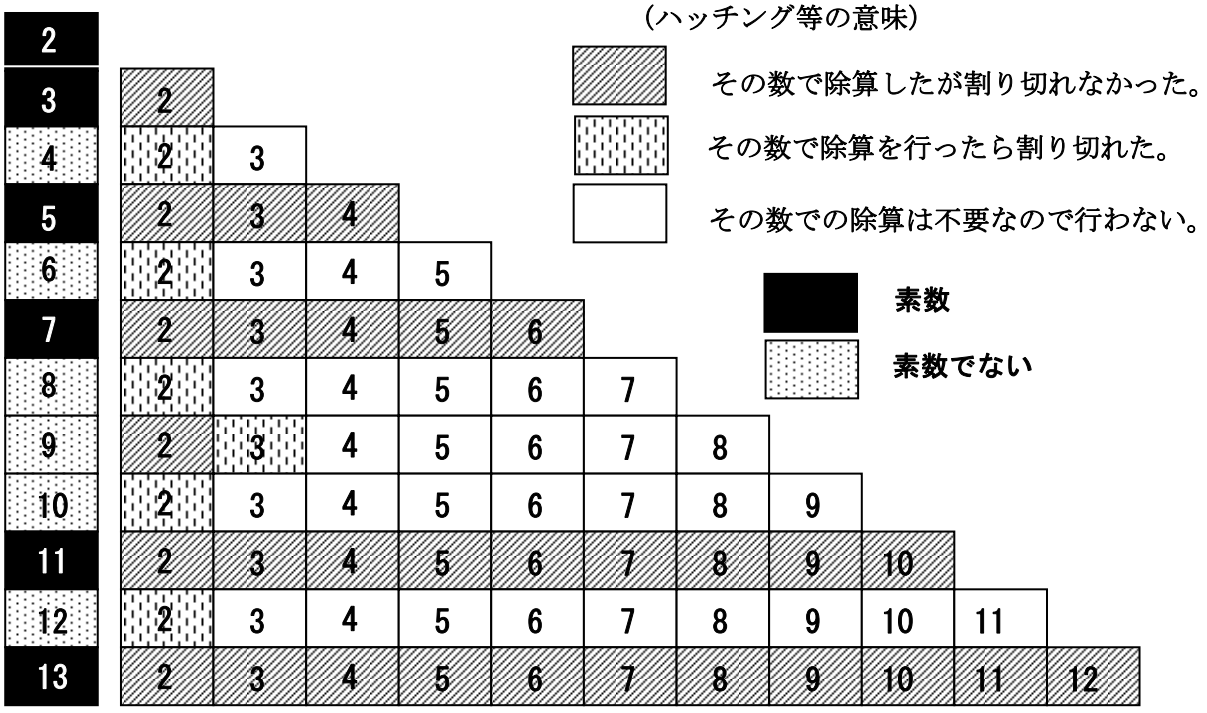
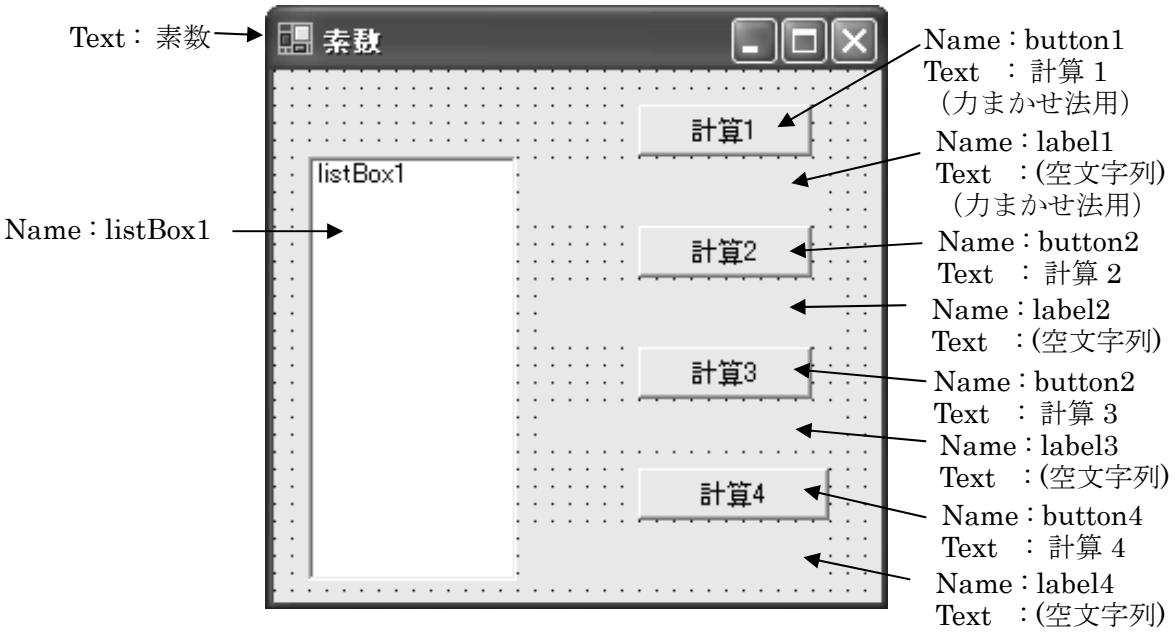


図 2-4　力まかせ法による素数の求め方

[Program 2-7] 力まかせ法による素数

力まかせ法だけではなく他の方法との比較のために以下のように Form 上に配置します。




```
private int   素数の数 = 0;
private int[] 素数     = new int[500];
private long 素数計算１() //力まかせの方法
{
  int i,n; long 計算数 =0; 素数の数=0;
  for(n = 2; n <= 1000; n++)
  {  for (i = 2; i < n; i++)
     { 計算数++;                    // 計算回数のカウント
       if ((n % i) == 0) break; // 余りが０であれば割り切れた
     }
     if (n == i)素数[素数の数++] = n; // 最後まで割り切れ
  }                                    // なければ素数とみなす
  return 計算数;
}
private void 表示(Label label, ListBox list,long X)
{
  int i; label.Text="割り算回数=" + X.ToString("#,##0");
  list.Items.Clear();
  for(i = 0; i < 素数の数; i++)
          list.Items.Add(素数[i].ToString("#,##0"));
}
private void button1_Click(object sender, System.EventArgs e)
{
  表示(label1, listBox1, 素数計算１());
}
```

(b) 改良1

2や3で割り切れなければ，4（＝ 2×2）や6（＝ 2×3）でも割り切れません。しかし，力まかせ法では，4や6でも割り切れるかどうかを判定しています。これは無駄です。そこで，求まった素数を配列に格納しておき，既に求まった素数で割り切れるかどうかを判定することにします。

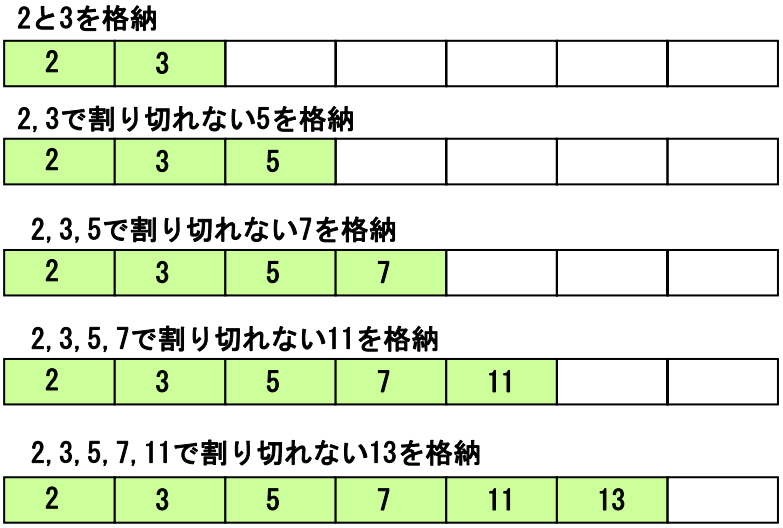


図 2-5　素数計算の改良 1

[Program 2-8] 素数計算の改良 1(表示は力まかせ法と同じ)

下線部が力まかせ法と異なる部分です。

```
private long 素数計算2() //改良1
{ int i,n; long 計算数 =0; 素数の数=0; //計算回数
  素数[素数の数++] = 2; 素数[素数の数++] = 3;
  for(n = 5; n <= 1000; n += 2)
  { bool flag = true;
    for (i = 1; i < 素数の数; i++){
      計算数++;                          // 計算回数のカウント
      if ((n % 素数[i]) == 0)            // 余りが0であれば
          {flag = false; break;}         // 割り切れた
    }
    if (flag)素数[素数の数++] = n; // 最後まで割り切れな
  }                                      // ければ素数とみなす
  return 計算数;
}
private void button2_Click(object sender, System.EventArgs e)
{   表示(label2, listBox1, 素数計算2());
}
```

(c) 改良 2

今, 100 の約数を考えてみると, 次のように割り切れる数のすべてが, 最初の 4 通りに入っています。すなわち，最初の 4 通りで割り切れなければ，素数として判別してもよいことになります。

2×50
4×25
5×20
10×10
20× 5
25× 4
50× 2

割り切れる数の
すべてが
この範囲内に入る。

⇒ Nの平方根以下の素数で
割り切れなければ,
素数として判別してよい。

これらの関係を図 2-6 に示すと，乗数と被乗数の組合せは，対称になっていることが分かります。すなわち，N の平方根以下の素数で割り切れなければ，素数とみなしてよいことが分かります。

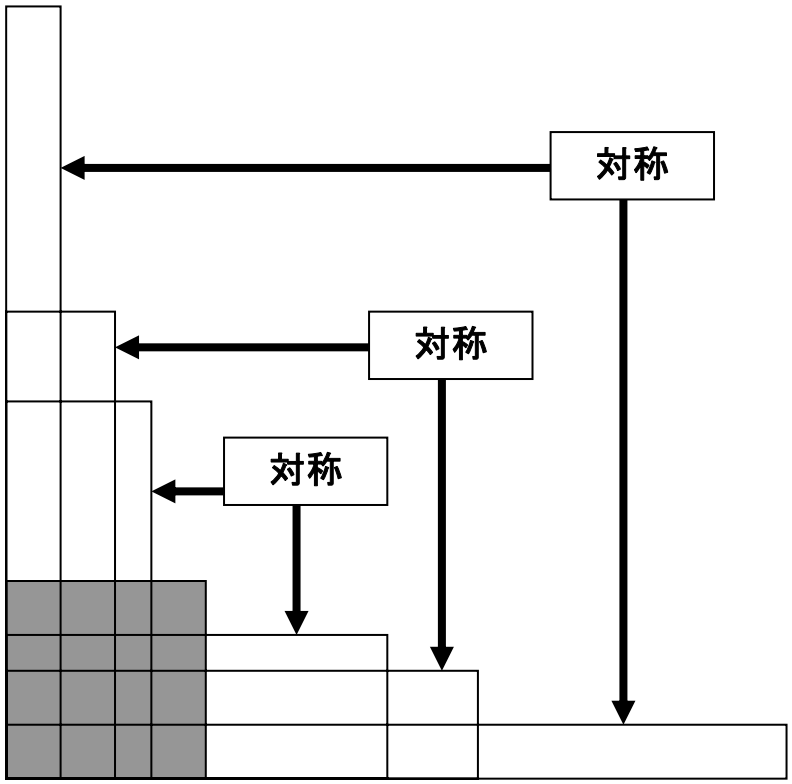


図 2-6　約数はお互いに対称

[Program 2-9] 素数計算の改良 2(表示は力まかせ法と同じ)

下線部が改良 1 と異なる部分です。

```
private long 素数計算3()       //改良2
{
  int i,n;  long 計算数 =0; 素数の数 = 0;//計算回数と素数の数
  素数[素数の数++] = 2; 素数[素数の数++] = 3;
  for(n = 5; n <= 1000; n += 2)
  { bool flag = true;
    for (i = 1; 素数[i] * 素数[i] <= n; i++)
    { 計算数++;                          // 計算回数のカウント
      if ((n % 素数[i]) == 0)            // 余りが0であれば
         { flag = false; break;}         // 割り切れた
    }
    if (flag) 素数[素数の数++] = n; // 最後まで割り切れな
  }                                     // ければ素数とみなす
  return 計算数;
}
private void button3_Click(object sender, System.EventArgs e)
{
   表示(label3, listBox1, 素数計算3());
}
```

(d) 改良 3(エラトステネスのふるい)

十分なメモリ量があれば，整数をすべて書き並べ，素数の倍数を消していくことで素数を求めることができます。この方法は，一切除算を使いませんので，計算効率のよい方法です。ただし，100 桁以上の素数を求めるには，通常メモリ量が不足しますので不都合かもしれません。

この方法は，提唱者の名前から**エラトステネスのふるい**(sieve of Eratosthenes)と呼ばれます。

① 2 以上 N 以下の整数をすべて書き並べておく。
② 2 の倍数をすべて消す。
③ 残った最小の数(最初は 3)の倍数をすべて消す。
④ 上記③を繰り返す。

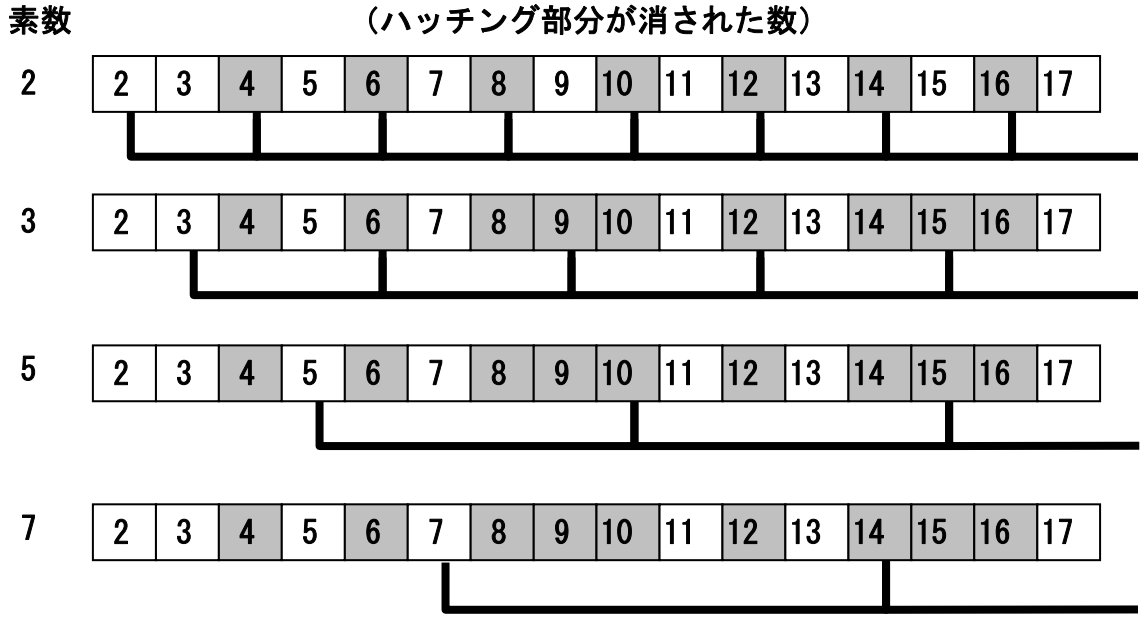


図 2-7 エラトステネスのふるい

[Program 2-10] エラトステネスのふるい（表示は力まかせ法と同じ）

ふるい用の配列を用意しておきます。

```
private int[] ふるい=new int[1001];
 .
 .
 .
private long 素数計算4() //エラトステネスのふるい
{   //  割り算を一切行わない方法
    int i,j;
    for(i = 2; i <= 1000; i++) ふるい[i]=i;
    for(i = 2; i <= 1000; i++)
      if(ふるい[i] != 0)for(j = 2 * i;j <= 1000; j += i) ふるい[j]=0;
    素数の数=0;
    for(i = 2;i <= 1000;i++)
        if(ふるい[i] != 0)素数[素数の数++] = ふるい[i];
    return 0;
}
private void button4_Click(object sender, System.EventArgs e)
{  表示(label4, listBox1, 素数計算4());
}
```

(5)文字列は文字型の配列

文字列は，16 ビット Unicode 文字の配列として表現されています。この点は従来の C++や C 言語と同じです。ただし文字列の最後を検出するのに整数(int)の 0 かどうかで判別することはできません。

例えば，文字数をカウントするのに

```
string a="abcd";
int num=0;
while(a[num]!=0) num++;
```

のようにすると,従来の C++や C 言語では a[num]!=0 が False となる，つまり a[num]==0 となる文字が見つかりますが, C#では実行時に配列添え字の例外となります。C#で文字列の長さを判別するには，プロパティ Length を使うか，以下のように，文字列を char 型の配列として宣言しましょう。

```
char []a= {'a','b','c','d','¥0'};
int num=0;
while(a[num] !=0) num++;
```

C#で次のように記述すると，下線部が「アクセスできない保護レベル」としてビルドエラーになってしまいます。

```
string X="abcde";
x[0]='d';
```

このときは，従来の C++や C 言語と同様，たとえば次のように記述しましょう。

```
char []x= {'a','b','c','d','¥0'};
x[0]='d';
```

以上のように，C#における文字列は，文字の配列という点では C++や C 言語と一致していますが，保護レベルや文字列サイズ等の取り扱いが異なりますので要注意です。

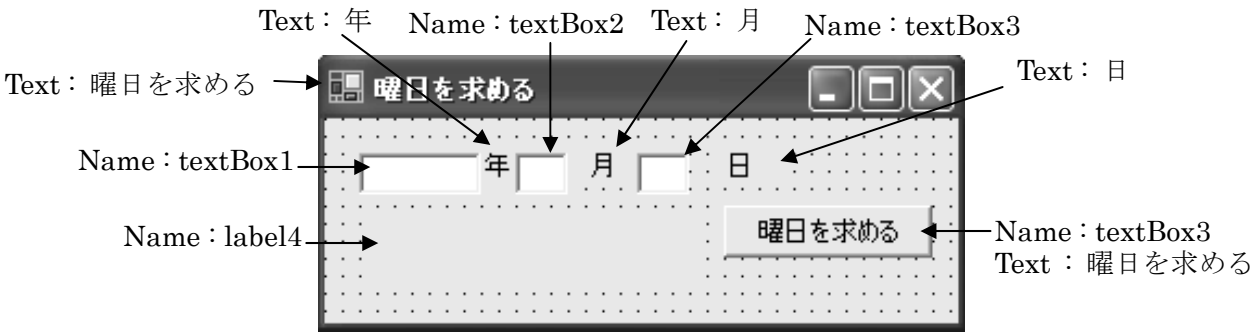
(6)ツェラーの公式

文字列を文字型の配列として使う例を示します。Program 2-11 は，ツェラー(Zeller)の公式で得られた曜日の番号を配列の添え字として，曜日を表示します。

番号	0	1	2	3	4	5	6
曜日	日	月	火	水	木	金	土

まず，曜日の文字列を string で指定しておき，曜日の番号をインデックスとして文字型のデータを取り出しています。

[Program 2-11] ツェラーの公式




```
// ツェラー(Zeller)の公式
private int WeekDay(int Y, int M, int D)
{ int YY = Y; int MM = M; if (M < 3){YY = Y - 1; MM = M + 12;}
  return (YY +(YY/4)-(YY/100)+(YY/400)+(13*MM + 8)/5+D) % 7;
}
private void button1_Click(object sender, System.EventArgs e)
{ int    Y = int.Parse(textBox1.Text);
  int    M = int.Parse(textBox2.Text);
  int    D = int.Parse(textBox3.Text);
  int    R = WeekDay(Y,M,D);
  string X = "日月火水木金土";
  label4.Text = X[R]+"曜日";
}
```