ポインタの復習とメモリの動的確保

榎本　理恵

講義内容

回	講義・実験内容
1	クラスの復習1
2	クラスの復習2(オペレータのオーバーロード)
3	ポインタの復習とメモリの動的確保
4	スタック
5	キュー
6	連結リスト①
7	連結リスト②(再帰)
8	二分木
9	中間テスト
10	ハッシュ
11	クイックソート
12	マージソート
13	ヒープソート
14	文字列探索
	学期末テスト

本日お話する内容

1. ポインタの復習
2. 関数の配列渡し
3. 二次元配列(宣言による二次元配列と
   配列へのポインタ配列による二次元配列の違い)
4. メモリの動的確保

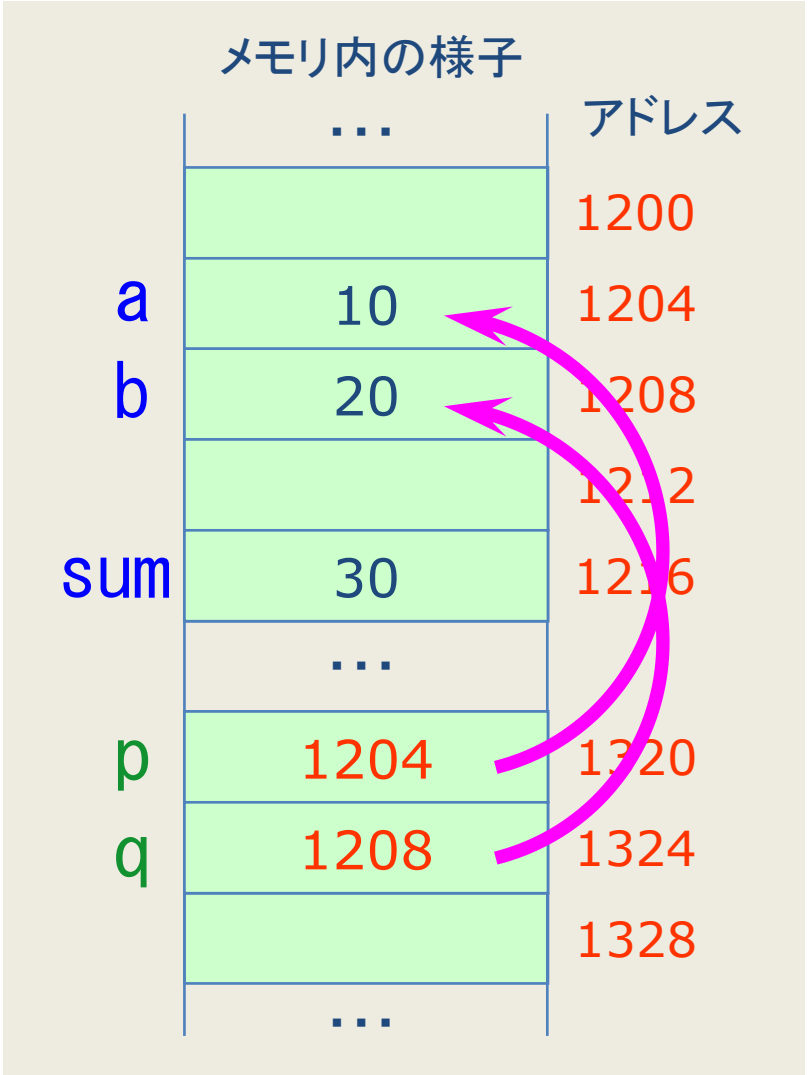
ポインタの復習

ポインタはアドレスを代入するための変数

ポインタを使用したコード例

```
int a=10, b=20;
int *p,  *q;

p=&a;
q=&b;

int sum=*p+*q;
```

ポインタはアドレスを代入して初めて使える



配列とアドレス

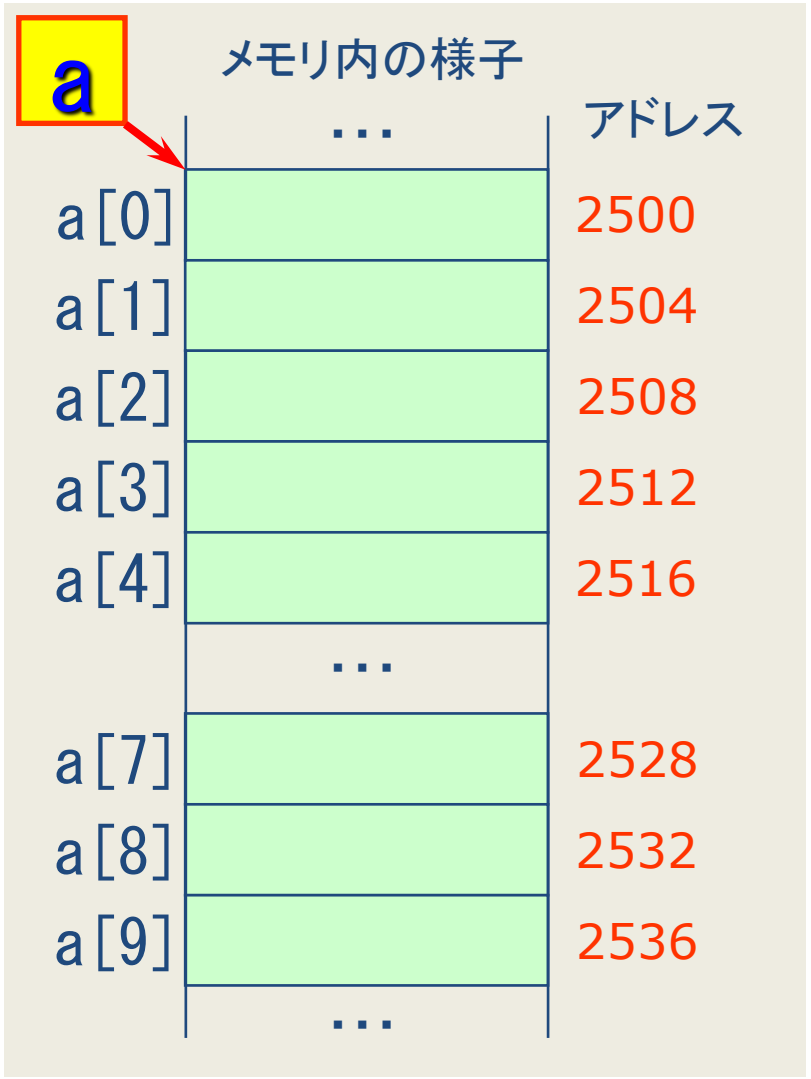
配列データは、同じ型のデータをメモリ内に連続して並べられる。

```
int a[10];
```

配列名は、
その領域の先頭アドレスを指す
アドレス定数である

ポインタと似ているが、
配列名自体は変数ではないので、
ポインタのように他のアドレスを
代入し直すことはできない。

上記配列aの先頭アドレスは、
　a
　&a[0]
のどちらでも表せる。



配列とポインタ①

ポインタを用いて配列データの各要素に値を代入する方法

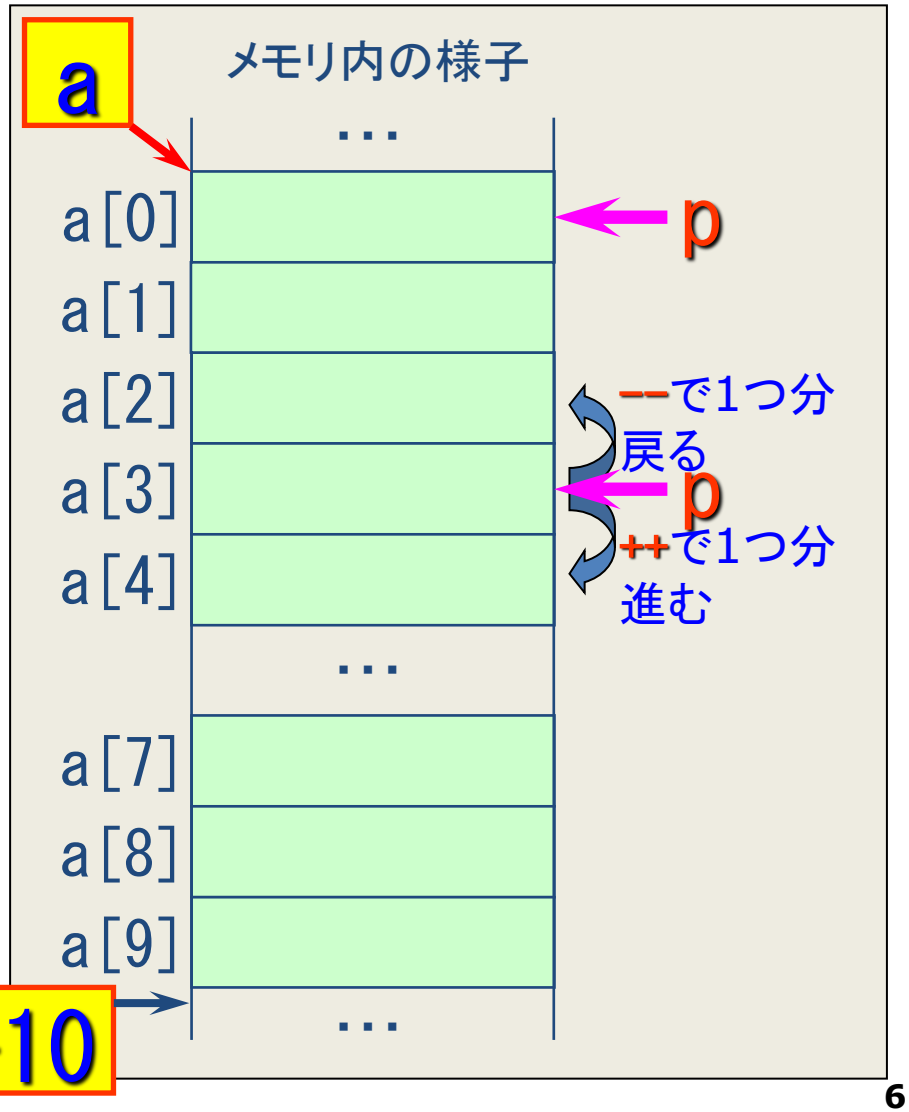
```
int a[10];
int *p;
```

```
for(int i=0; i<10; i++)
  cin >> a[i];
```

同じ処理を意味する

```
for(p=a; p<a+10; p++)
  cin >> *p;
```

ポインタにとって++や--は指す位置を、
データ型1つ分ずらすことを意味する。
アドレスの値を1増減する意味ではない。

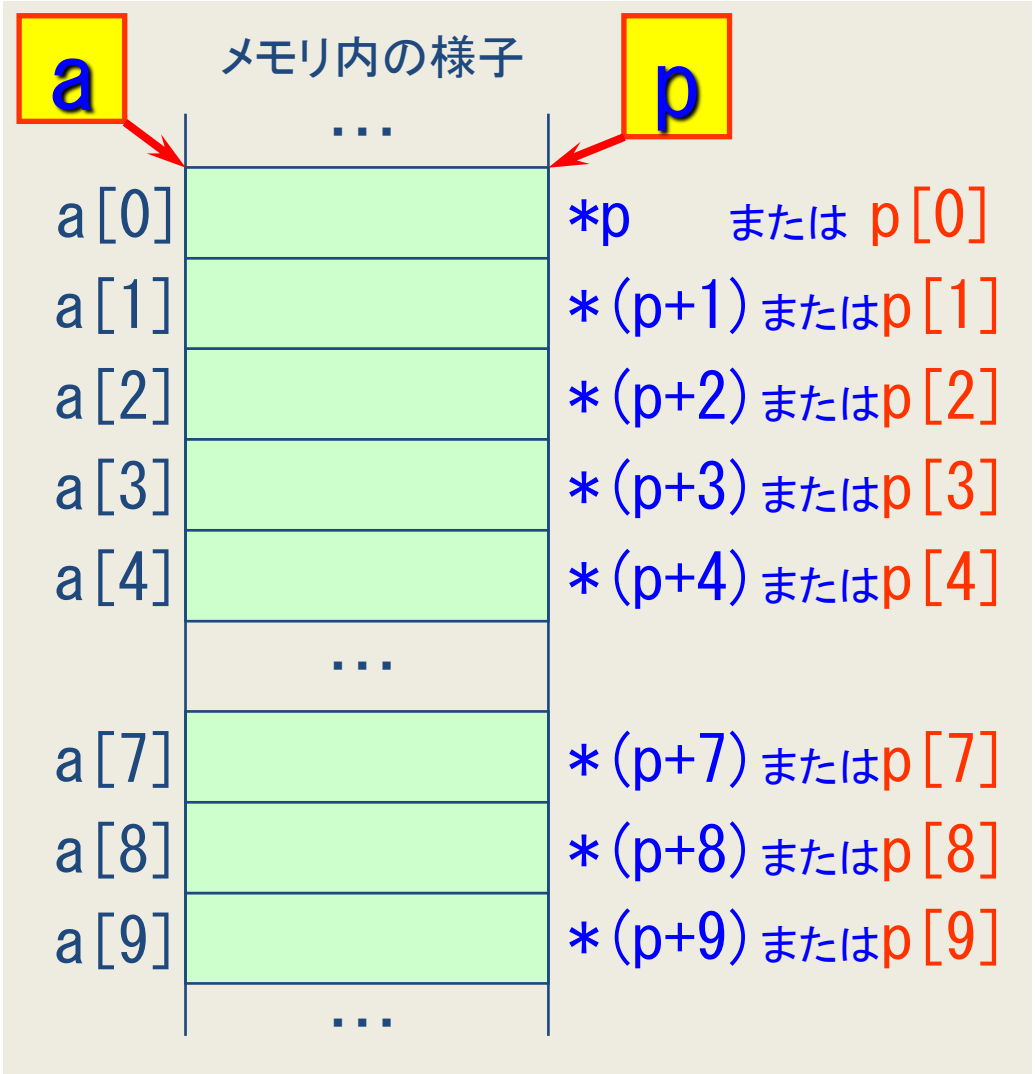


配列とポインタ②

ポインタを用いて配列データの各要素に値を代入する方法

```
int a[10];
int *p=a;
 // pを配列aの先頭アドレスで
 // 初期化

a[2]=38;
*(p+2)=38;
p[2]=38;
```

どれも同じ代入

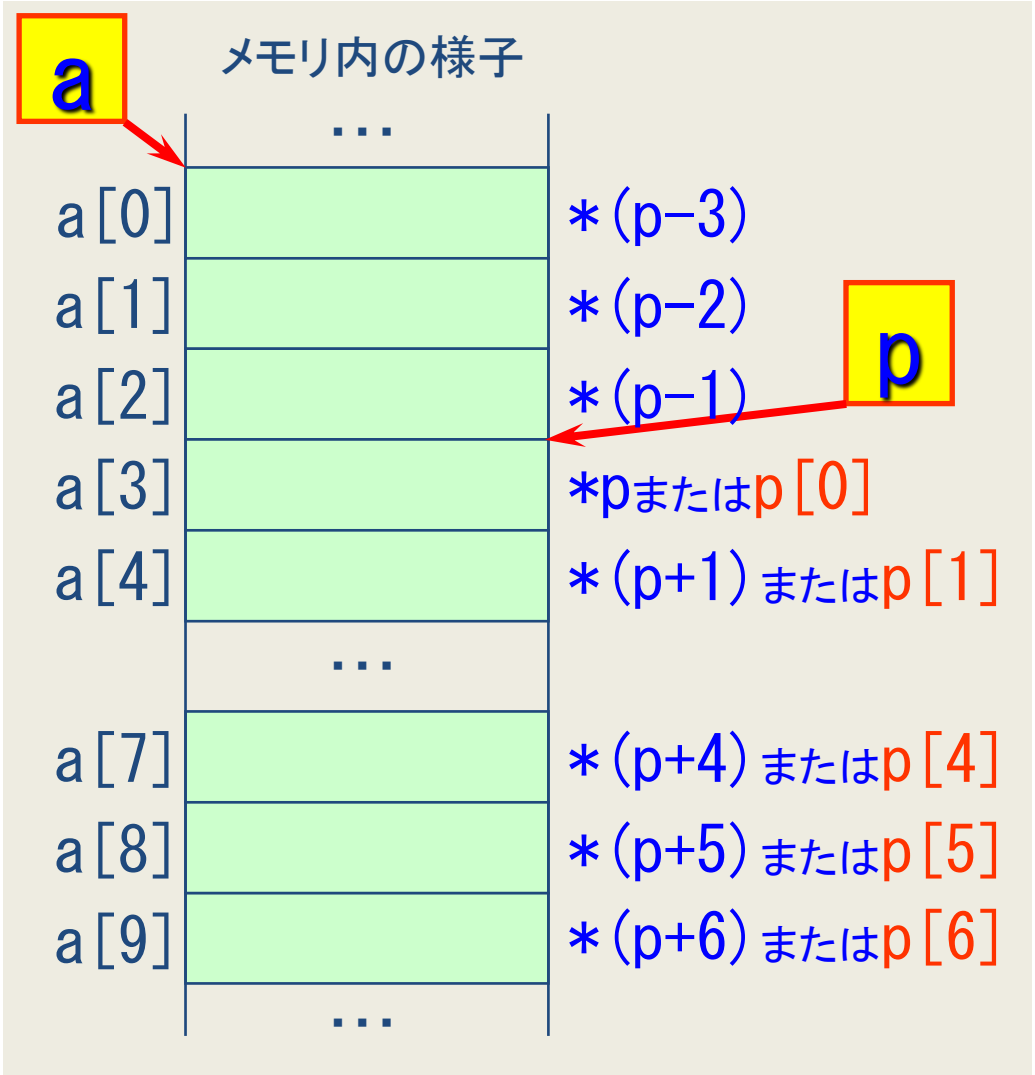


配列とポインタ③

ポインタを用いて配列データの各要素に値を代入する方法

```
int a[10];
int *p=&a[3];
 // pをa[3]のアドレスで
 // 初期化

a[2]=38;
*(p-1)=38;
```

どちらも同じ代入



ポインタ間の演算

◎同じ型の2つのポインタの差は定義されている

```
int a[8];
int *p1=a, *p2=&a[5];
```

このあと、例えば p2-p1 という計算は可能

◎ポインタ間の差は整数となる

- ポインタの指すオブジェクト単位で差を表す
- 何要素分ずれているかを示す
- 結果が負になることもある
- 同一配列内の要素へのポインタの差のみ計算は有効

上記の例で、p2-p1の値は5となる

◎2つのポインタの和は計算できない

```
cout << (p1 + p2);
```

←コンパイルエラー

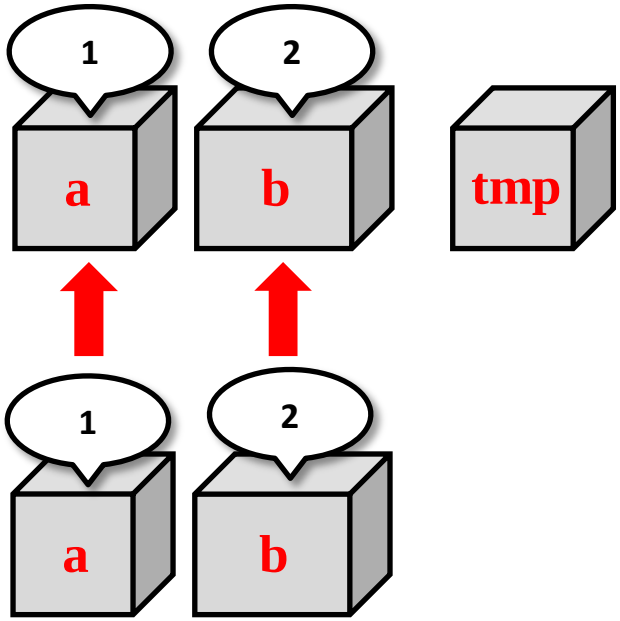
本日お話する内容

1. ポインタの復習
2. 関数の配列渡し
3. 二次元配列(宣言による二次元配列と
   配列へのポインタ配列による二次元配列の違い)
4. メモリの動的確保

関数の値渡し

```
#include <iostream>
using namespace std;

void irekae(int a, int b){
    int tmp;

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

    cout << "a=" << a << ",b=" << b << "¥n";
    irekae(a, b);
    cout << "a=" << a << ",b=" << b << "¥n";
    return 0;
}
```

本プログラムを実行した際の出力を書きなさい。



■ 実行例

```
comsv% ./out
a=1,b=2
a=1,b=2
comsv%
```

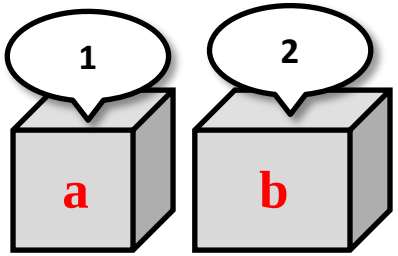
関数の参照渡し

```
#include <iostream>
using namespace std;

void irekae(int &a, int &b){
    int tmp;

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

    cout << "a=" << a << ",b=" << b << "¥n";
    irekae(a, b);
    cout << "a=" << a << ",b=" << b << "¥n";
    return 0;
}
```

関数を使って、main側の変数の値を入れ替えることはできないか？

⇒関数に箱そのものを渡せばよい

⇒引数の変数の前に&をつける



関数に、
箱そのものをわたす。

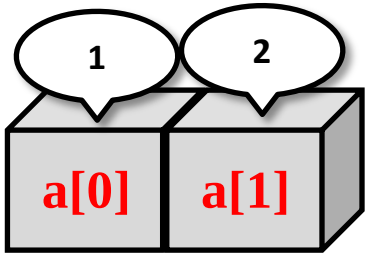
■ 実行例

```
comsv% ./out
a=1,b=2
a=2,b=1
comsv%
```

関数の配列渡し①

```
#include <iostream>
using namespace std;

void irekae(int a[], int num){
    int tmp;
    for(int i=0;i<num/2;i++){
        tmp=a[i];
        a[i]=a[num-i-1];
        a[num-i-1]=tmp;
    }
}
int main(){
    int a[2]; a[0]=1; a[1]=2;

    cout << "a[0]=" << a[0] << ",a[1]=" << a[1] << "¥n";
    irekae(a, 2);
    cout << "a[0]=" << a[0] << ",a[1]=" << a[1] << "¥n";
    return 0;
}
```

配列の中身を逆順にするプログラム
⇒関数に配列の先頭アドレスを渡せばよい
（但し、関数側で配列の値を操作すると、
呼び出し側の配列に反映される）



関数に、
配列の先頭アドレスをわたす。

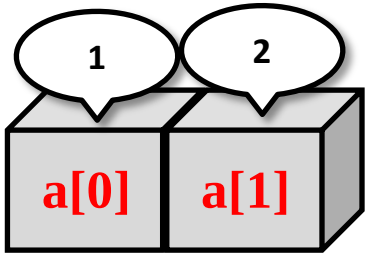
■ 実行例

```
comsv% ./out
a[0]=1,a[1]=2
a[0]=2,a[1]=1
comsv%
```

関数の配列渡し②

```
#include <iostream>
using namespace std;

void irekae(int *a, int num){
    int tmp;
    for(int i=0;i<num/2;i++){
        tmp=a[i];
        a[i]=a[num-i-1];
        a[num-i-1]=tmp;
    }
}
int main(){
    int a[2]; a[0]=1; a[1]=2;

    cout << "a[0]=" << a[0] << ",a[1]=" << a[1] << "¥n";
    irekae(a, 2);
    cout << "a[0]=" << a[0] << ",a[1]=" << a[1] << "¥n";
    return 0;
}
```

先ほどと同じ解釈になる別の書き方のプログラム例



関数に、
配列の先頭アドレスをわたす。

■ 実行例

```
comsv% ./out
a[0]=1,a[1]=2
a[0]=2,a[1]=1
comsv%
```

本日お話する内容

1. ポインタの復習
2. 関数の配列渡し
3. 二次元配列(宣言による二次元配列と
   配列へのポインタ配列による二次元配列の違い)
4. メモリの動的確保

宣言による二次元配列①

```
int a[2][3];
```

以下はあくまでも概念上のもの

a[0][0]	a[0][1]	a[0][2]
a[1][0]	a[1][1]	a[1][2]

上記の宣言で二次元配列を確保しても、
実際にメモリ上に二次元で確保するわけではないので、
注意が必要である。
(a[2][3]=5;のように値を代入し、
cout << a[2][3] <<"¥n";のように値を読み出している分には
問題は生じない)

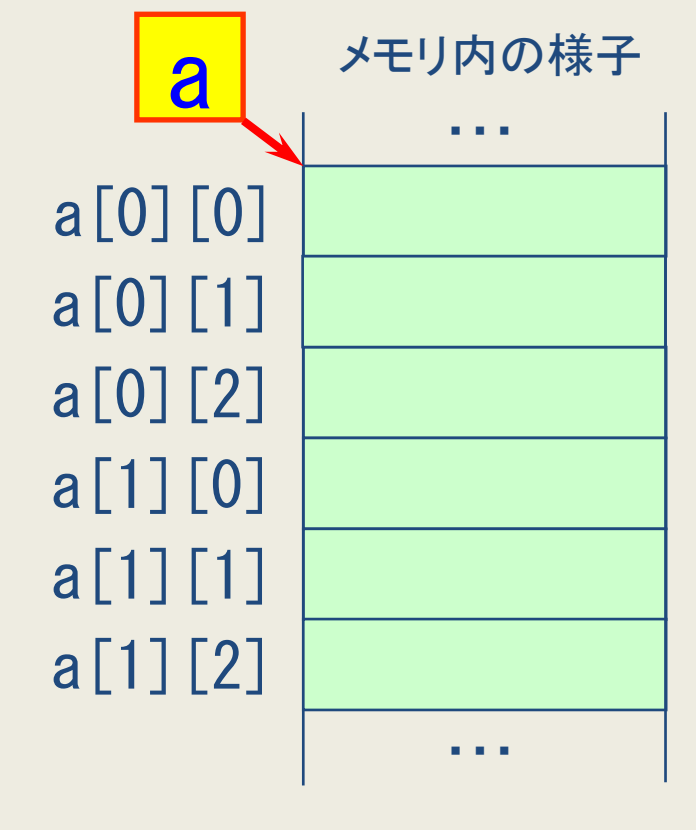
宣言による二次元配列②

概念上は二次元配列であるが、実際には一次元配列である

```
int a[2][3];
```

a[0]は&(a[0][0])と同じものである。
a[1]は&(a[1][0])と同じものである。

a と a[0] と &a[0][0] はどれも同じアドレスを持つ。
ただし、型が異なる。
a は int [][]またはint**、
a[0]と &a[0][0]は int[]またはint*

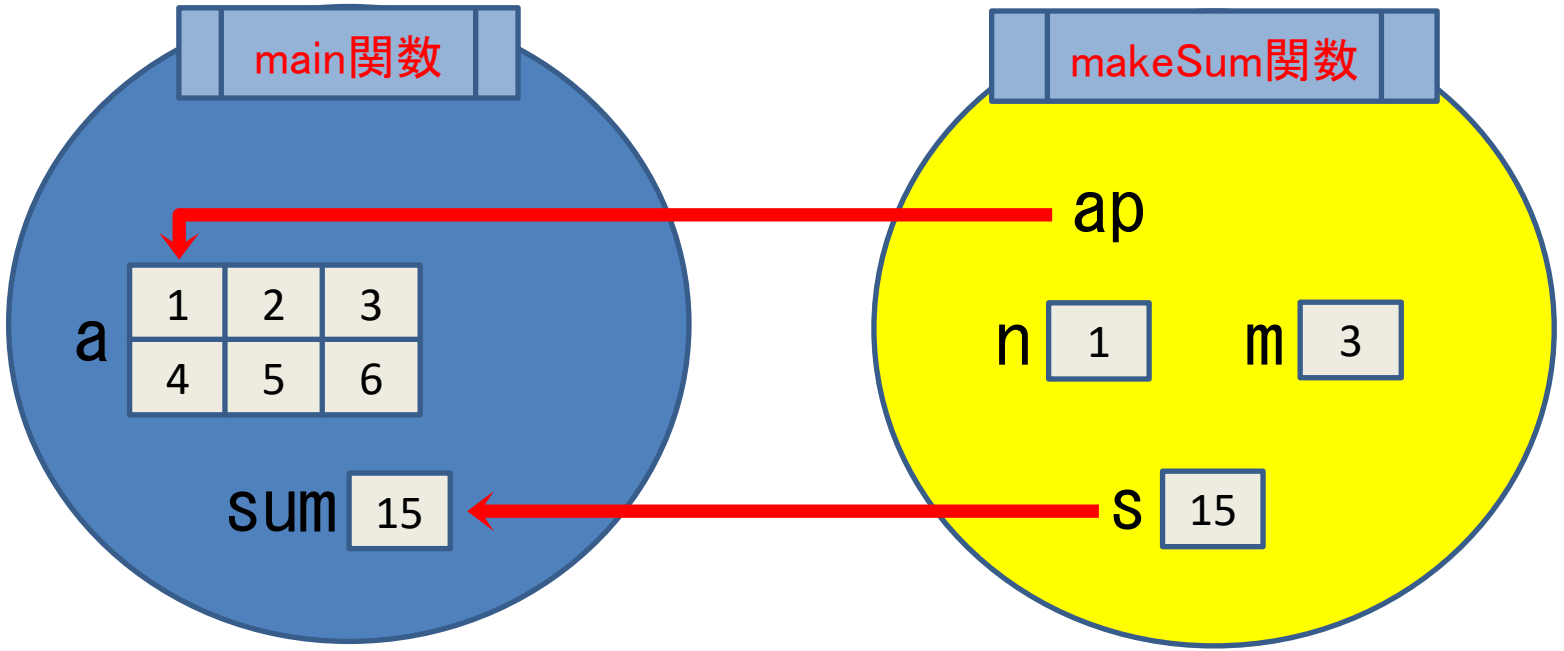


宣言による二次元配列③

配列のn行目の総和を返す関数への値の渡し方

```
int main(){
    int a[2][3]={{1,2,3},
                 {4,5,6}};
    int sum;
    sum=makeSum(a,1,3);
    …
```

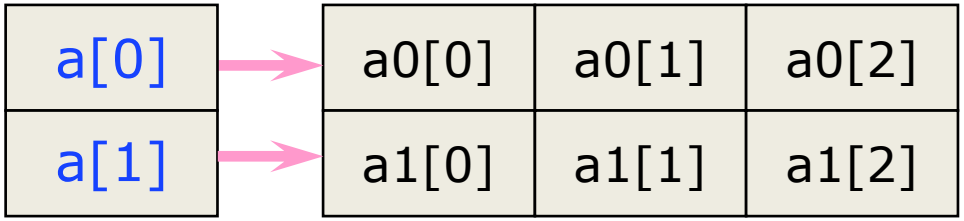
```
int makeSum(int ap[][3], int n, int m){
    int s=0;
    for(int i=0; i<m; i++){s+=ap[n][i];}
    return s;
}
```



配列へのポインタ配列による二次元配列①

```
int a0[3];
int a1[3];
int *a[2] = {a0, a1};
```

メモリ内の様子



a[0]には、a0[0]のアドレスが代入されている。
a[1]には、a1[0]のアドレスが代入されている。

使い方は、基本的にint a[2][3]と同じで、

a[1][2]=5;のように値を代入し、

cout << a[1][2] <<"¥n";のように値を読み出せばよい。

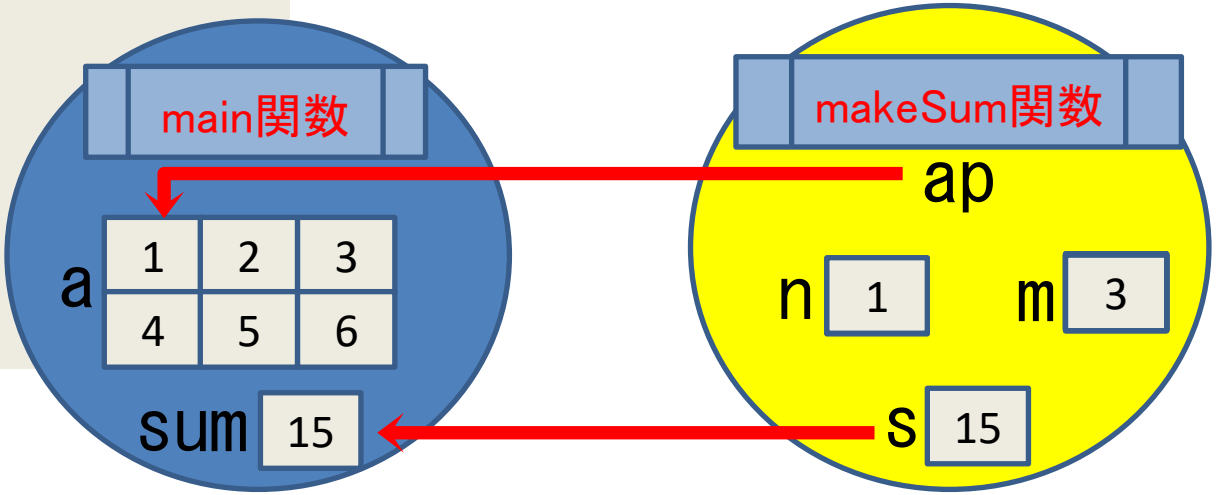
配列へのポインタ配列による二次元配列②

配列のn行目の総和を返す関数への値の渡し方

```
int main(){
    int a0[3];      int a1[3];
    int *a[2] = {a0, a1};
    int p=1;
    for(int i=0;i<2;i++){
        for(int j=0;j<3;j++){
            a[i][j]=p;
            p++;
        }
    }
    int sum;
    sum=makeSum(a,1,3);
    ....
```

```
int makeSum(int **ap, int n, int m){
    int s=0;
    for(int i=0; i<m; i++)  s+=ap[n][i];
    return s;
}
```

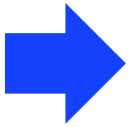
makeSumの第1引数において、
配列の大きさを指定する必要がなくなる



配列へのポインタ配列による二次元配列③

あらかじめ行と列の数が分かっていれば、左側のように
配列へのポインタ配列を構築することが可能であるが、
現実的には、データサイズにより行と列の数は変わってくる。
このため、実際には動的にメモリを確保することにより、
配列へのポインタ配列を構築することになる。

```
int a0[3];
int a1[3];
int *a[2] = {a0, a1};
```



```
int u=2;      int v=3;
int **a=new int*[u];
for(int i=0;i<u;i++){
        a[i]=new int[v];
}
```

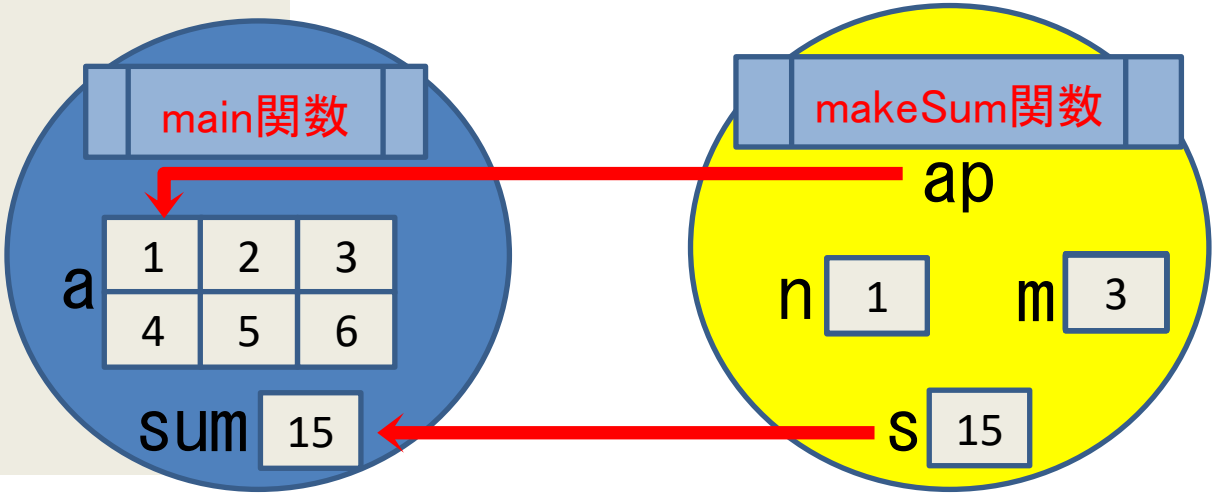
配列へのポインタ配列による二次元配列④

配列のm行目の総和を返す関数への値の渡し方

```
int main(){
    int u=2;          int v=3;
    int **a=new int*[u];
    for(int i=0;i<u;i++){
        a[i]=new int[v];
    }
    int p=1;
    for(int i=0;i<2;i++){
        for(int j=0;j<3;j++){
            a[i][j]=p;      p++;
        }
    }
    int sum;
    sum=makeSum(a,1,3);
    ....
```

```
int makeSum(int **ap, int n, int m){
    int s=0;
    for(int i=0; i<m; i++){s+=ap[n][i];}
    return s;
}
```

本プログラムの場合、行列の大きさは、
例えば標準入力で入力された値でも可能



本日お話する内容

1. ポインタの復習
2. 関数の配列渡し
3. 二次元配列(宣言による二次元配列と
   配列へのポインタ配列による二次元配列の違い)
4. メモリの動的確保

単語リストを配列に読み込むプログラム例

あらかじめ単語がいくつあるのか想像し、
その数以上で配列を用意する必要がある。
(現実的に、ユーザーが用意する単語が
いくつあるのか事前に分かることは
ありえない！)
⇒その場で動的に配列を確保すればよい

```
int main() {
        string word[100];   // 単語の配列
        string listfilename="wordslist.txt";

        ifstream listfin(listfilename);
        if(!listfin){
            cerr << "ERROR:ファイル:" << listfilename << "を開けません。" <<endl;
            exit(EXIT_FAILURE);
        }
        int wordcount=0;
        while(listfin >> word[wordcount]){
             wordcount++;
             if(wordcount>=100){
                  cerr << "wordの配列数を増やす必要があります" << endl;
                  exit(EXIT_FAILURE);
             }
        }
        cout << "単語数=" << wordcount << "¥n";
        return 0;
}
```

単語リストを配列に読み込むプログラム例(new版)

```
int main() {
        string listfilename="wordslist.txt";

        // メモリをとるために、ファイルの行数=単語の数をカウントする
        ifstream listfin(listfilename);
        if(!listfin){
            cerr << "ERROR:ファイル:" << listfilename << "を開けません。" <<endl;
            exit(EXIT_FAILURE);
        }
        string tmp;
        int wordcount=0;
        while(listfin >> tmp){
             wordcount++;
        }
        listfin.close();

        string *word;                          // 単語の変数へのポインタ
        word=new string[wordcount]; //メモリの動的確保

        //単語数をカウントし、メモリを確保したので、実際に値を代入する
        listfin.open(listfilename);
        if(!listfin){
            cerr << "ERROR:ファイル:" << listfilename << "を開けません。" <<endl;
            exit(EXIT_FAILURE);
        }
        int i=0;
        while(listfin >> word[i]){
             i++;
        }

        cout << "単語数=" << wordcount << "¥n";
        delete[] word;
        return 0;
}
```

単語リストを配列に読み込むプログラム例(余計版)

```
int main() {
        string listfilename="wordslist.txt";

        // メモリをとるために、ファイルの行数=単語の数をカウントする
        ifstream listfin(listfilename);
        if(!listfin){
            cerr << "ERROR:ファイル:" << listfilename << "を開けません。" <<endl;
            exit(EXIT_FAILURE);
        }
        string tmp;
        int wordcount=0;
        while(listfin >> tmp){
             wordcount++;
        }
        listfin.close();

        string word[wordcount];
        //実は最近のコンパイラではこれがとおるのです・・・但し、宣言でメモリをとると
        //開放はできませんので、その部分ではnew、deleteが勝っています

        //単語数をカウントし、メモリを確保したので、実際に値を代入する
        listfin.open(listfilename);
        if(!listfin){
            cerr << "ERROR:ファイル:" << listfilename << "を開けません。" <<endl;
            exit(EXIT_FAILURE);
        }
        int i=0;
        while(listfin >> word[i]){
             i++;
        }

        cout << "単語数=" << wordcount << "¥n";
        return 0;
}
```

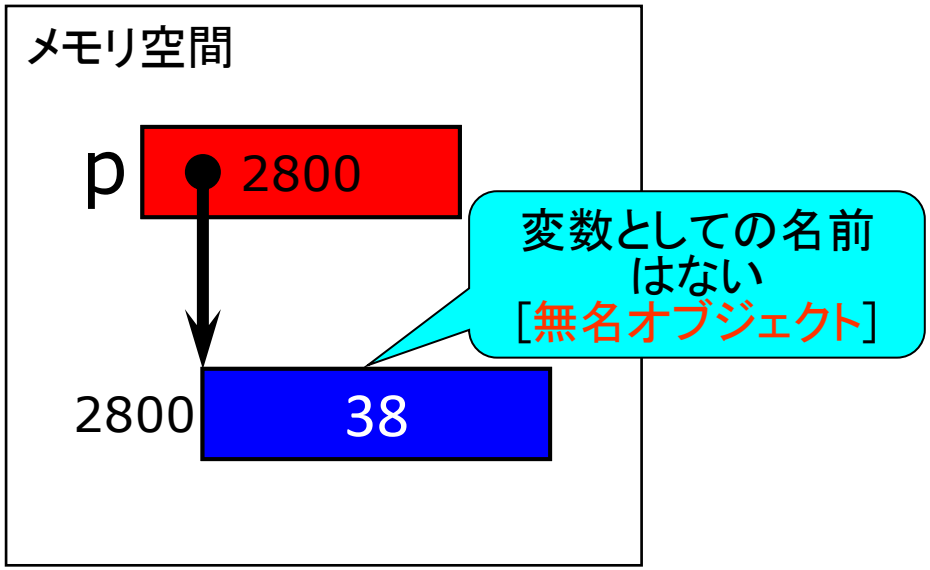
メモリの動的割り付けとは

newで確保し、deleteで解放する

例：int型の領域を1個確保して、あとで解放する。

```
int *p;
    ...
p=new int;
    ...
*p=10+4*7;
    ...
delete p;
```

int型の領域を確保



newで確保された領域は、明示的にdeleteされるまで存続する。
明示的にdeleteされなければ、プログラムの終了時まで存続する。

メモリの動的割り付け①

- newの使い方(1)

ポインタ＝**new** 型；　「ポインタ」の型は、newしたい型と同じ

例：

```
int *p=new int;
int *q=new int(15);  // 領域確保して15で初期化
int *p1=new int();   // 領域確保して0で初期化

string *sp1=new string;
                // 領域確保して空文字列で初期化
string *sp2=new string("School");
                // 領域確保してSchoolという文字列で初期化
string *sp2=new string();
                // 領域確保して空文字列で初期化
```

メモリの動的割り付け②

- newの使い方(2)　自作クラスをnewする

例：以前のComplexクラスの場合

```
class Complex {
    double  real, imaginary;   // 実数部realと虚数部imaginary
  public:
    Complex(double a=0, double b=0){  real=a;  imaginary=b; }
       ～以下略～
};
```

上のクラスが定義されているとき、
例えばmainの中で以下のようにメモリを確保することができる。

```
Complex *p=new Complex();
        // 領域確保し、コンストラクタで初期化　0+0i
Complex *q=new Complex(2,7);
        // 領域確保し、コンストラクタで初期化　2+7i
```

メモリの動的割り付け③

- newの使い方(3)　複数のメモリを動的に確保する

```
int *p=new int;              // 1個のメモリを確保
int *q=new int(15);          // 1個のメモリを確保して15で初期化
int *r=new int[15];          // 15個のメモリを確保

string *p=new string;        // 1個のメモリを確保
string *q=new string("School");
            // 1個のメモリを確保してSchoolという文字列で初期化
string *r=new string[15];
            // 15個のメモリを確保

Complex *p=new Complex();          // 1個のメモリを確保
Complex *q=new Complex(2,7);
            // 1個のメモリを確保して、2と7で初期化(2+7i)
Complex *r=new Complex[15];        // 15個のメモリを確保
```

メモリの解放

- deleteの使い方

delete ポインタ; （1個newした場合）

delete[] ポインタ; （複数個newした場合）

解放されるのがクラスオブジェクトの場合、
そのオブジェクトが消えることになるので、
クラスのデストラクタが実行されることになる。

【注】newで確保したメモリ以外は解放できない。

```
int a;
int *p=&a;
・・・
delete p;   // これはできない。
```

new-deleteの細かい注意点①

- new・mallocで割り当てられるのはヒープという領域
  宣言による配列で割り当てられるのはスタックという領域
  - これらの領域は通常サイズの上限値が設定されているので、
    最大で使いたい場合には、自分で設定する必要がある。
  - newでとられる領域を増やす場合、ヒープを増やし、
    宣言による配列でとられる領域を増やす場合、スタックを増やす必要がある。
  - unix系の場合には、「ulimit -a」で表示される際の、
    「data seg size」がヒープ領域の上限値、
    「stacksize」がスタック領域の上限値を表す。
  - 一般的に大きいメモリはヒープで確保する約束になっており、
    スタックの上限値は小さいため、配列好きな方は要注意です！

new-deleteの細かい注意点②

- mallocしたメモリはfreeで開放、newしたメモリはdeleteで開放

```
<mallocの例>
int *p;
p=(int *) malloc(sizeof(int)*15);  // 15個のメモリを確保
for(i=0;i<15;i++){
        p[i]=i;
}
free(p);
```

- したがって、人と一緒に開発する際には、
  誰がどのようにメモリをとるのか明確にしておく必要がある。

本日はここまでです
お疲れさまでした

質問があれば、下記までお願いします
11号館4階1405号室

付録

1. ポインタの復習
2. 関数の配列渡し
3. メモリの動的確保

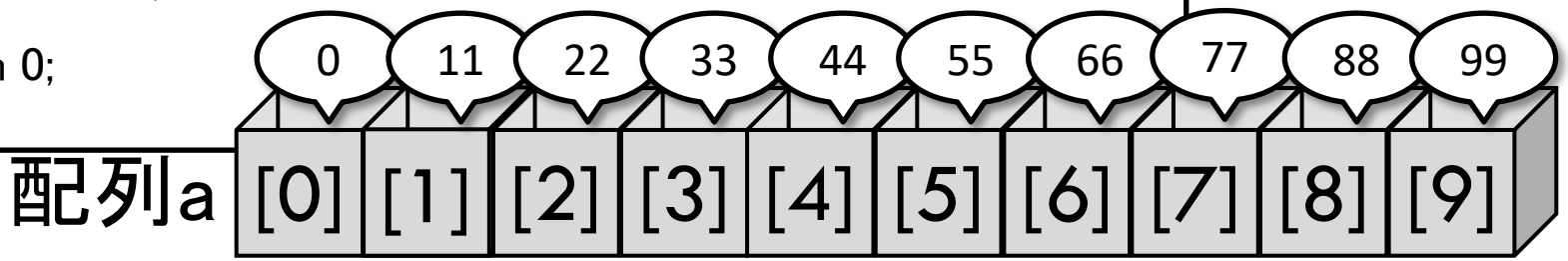
配列とポインタ練習問題

下記のプログラムの出力を書きなさい。

```
int main(){
    int a[10]={0,11,22,33,44,55,66,77,88,99};
    int *pointer1=a;
    int *pointer2=a+5;

    cout << "pointer1が指しているのは" << *pointer1 << "¥n";
    cout << "pointer2が指しているのは" << *pointer2 << "¥n";

    pointer1++;
    pointer2--;
    cout << "pointer1が指しているのは" << *pointer1 << "¥n";
    cout << "pointer2が指しているのは" << *pointer2 << "¥n";
    cout << "pointer2と1の差=" << pointer2-pointer1 << "¥n";

    pointer1=a+5;
    pointer2=pointer1-4;
    cout << "pointer2[3]が指しているのは" << pointer2[3] << "¥n";
    *(pointer2+2)=-1;
    for(int i=0;i<10;i++){ cout << " " << a[i]; }
    cout << "¥n";

    return 0;
}
```

■ 実行例

```
comsv% ./out
pointer1が指しているのは
pointer2が指しているのは
pointer1が指しているのは
pointer2が指しているのは
pointer2と1の差=
pointer2[3]が指しているのは

comsv%
```



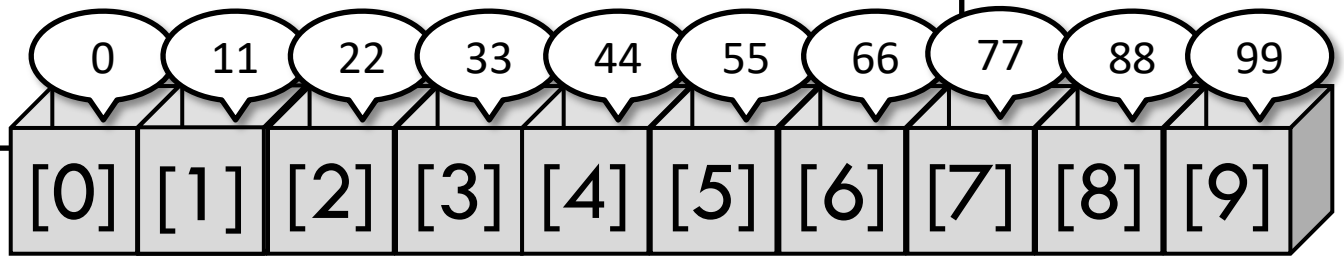
配列とポインタ練習問題解答

下記のプログラムの出力を書きなさい。

```
int main(){
    int a[10]={0,11,22,33,44,55,66,77,88,99};
    int *pointer1=a;
    int *pointer2=a+5;

    cout << "pointer1が指しているのは" << *pointer1 << "¥n";
    cout << "pointer2が指しているのは" << *pointer2 << "¥n";

    pointer1++;
    pointer2--;
    cout << "pointer1が指しているのは" << *pointer1 << "¥n";
    cout << "pointer2が指しているのは" << *pointer2 << "¥n";
    cout << "pointer2と1の差=" << pointer2-pointer1 << "¥n";

    pointer1=a+5;
    pointer2=pointer1-4;
    cout << "pointer2[3]が指しているのは" << pointer2[3] << "¥n";
    *(pointer2+2)=-1;
    for(int i=0;i<10;i++){ cout << " " << a[i]; }
    cout << "¥n";

    return 0;
}
```

■ 実行例

```
comsv% ./out
pointer1が指しているのは0
pointer2が指しているのは55
pointer1が指しているのは11
pointer2が指しているのは44
pointer2と1の差=3
pointer2[3]が指しているのは44
 0 11 22 -1 44 55 66 77 88 99
comsv%
```



付録

1. ポインタの復習
2. 関数の配列渡し
3. メモリの動的確保

練習問題

mainの配列scoreの総和を返す関数を4とおりの方法で作成しなさい

戻り値による方法

```
    makeSum(                                    ){




}
int main(){
    int score[5]={4,6,2,3,1};
    int sum;

    sum=makeSum(score,5);
    cout<<"配列の合計"<<sum<<"¥n";
    return 0;
}
```

参照変数による方法

```
void makeSum(                                   ){




}
int main(){
    int score[5]={4,6,2,3,1};
    int sum;

    makeSum(score,5,sum);
    cout<<"配列の合計"<<sum<<"¥n";
    return 0;
}
```

練習問題(続き)

mainの配列scoreの総和を返す関数を4とおりの方法で作成しなさい

アドレス渡しによる方法(関数内は配列利用)

```
void makeSum(int *pointer,                    ){




}
int main(){
    int score[5]={4,6,2,3,1};
    int sum;

    makeSum(score,5,&sum);
    cout<<"配列の合計"<<sum<<"¥n";
    return 0;
}
```

アドレス渡しによる方法(関数内はポインタ利用)

```
void makeSum(int *pointer,                    ){




}
int main(){
    int score[5]={4,6,2,3,1};
    int sum;

    makeSum(score,5,&sum);
    cout<<"配列の合計"<<sum<<"¥n";
    return 0;
}
```

練習問題解答

mainの配列scoreの総和を返す関数を4とおりの方法で作成しなさい

戻り値による方法

```
int makeSum(int hairetsu[], int kosuu){
    int all=0;
    for(int i=0;i<kosuu;i++){
        all+=hairetsu[i];
    }
    return all;
}
int main(){
    int score[5]={4,6,2,3,1};
    int sum;

    sum=makeSum(score,5);
    cout<<"配列の合計"<<sum<<"¥n";

    return 0;
}
```

参照変数による方法

```
void makeSum(int hairetsu[], int kosuu, int& all){
    all=0;
    for(int i=0; i<kosuu; i++){
        all+=hairetsu[i];
    }
}

int main(){
    int score[5]={4,6,2,3,1};
    int sum;

    makeSum(score,5,sum);
    cout<<"配列の合計"<<sum<<"¥n";

    return 0;
}
```

練習問題解答(続き)

mainの配列scoreの総和を返す関数を4とおりの方法で作成しなさい

アドレス渡しによる方法(関数内は配列利用)

```
void makeSum(int *pointer, int kosuu, int *all){
    *all=0;
    for(int i=0; i<kosuu; i++){
        *all+=pointer[i];
    }
}
int main(){
    int score[5]={4,6,2,3,1};
    int sum;

    makeSum(score,5,&sum);
    cout<<"配列の合計"<<sum<<"¥n";

    return 0;
}
```

アドレス渡しによる方法(関数内はポインタ利用)

```
void makeSum(int *pointer, int kosuu, int *all){
    *all=0;
    for(int *p=pointer; p<(pointer+kosuu); p++){
        *all+=*p;
    }
}
int main(){
    int score[5]={4,6,2,3,1};
    int sum;

    makeSum(score,5,&sum);
    cout<<"配列の合計"<<sum<<"¥n";

    return 0;
}
```

付録

1. ポインタの復習
2. 関数の配列渡し
3. メモリの動的確保

練習問題

標準入力から入力された得点の平均を求めるプログラムを作りなさい

```
int main(){
        int maxsize,count;
        cout << "平均点を計算します。入力するデータ数を入れてください¥n";
        cin >> maxsize;
```

実行例

```
平均点を計算します。入力する
データ数を入れてください
5
10
20
30
40
50
入力されたデータ数に達しました
データ数=5,平均点=30
```

```
        cout << "データ数=" << count << ",平均点=" << average << "¥n";
        return 0;
}
```

練習問題解答

標準入力から入力された得点の平均を求めるプログラムを作りなさい

```
int main(){
        int maxsize,count;
        cout << "平均点を計算します。入力するデータ数を入れてください¥n";
        cin >> maxsize;
       int *tokuten=new int[maxsize];
       count=0;
       while(cin >> tokuten[count]){
            count++;
            if(count>=maxsize){
                cout << "入力されたデータ数に達しました¥n";
                break;
            }
       }
       double average=0.0;
       for(int i=0;i<count;i++){
           average+=tokuten[i];
       }
       average/=count;
       delete[] tokuten;
       cout << "データ数=" << count << ",平均点=" << average << "¥n";
       return 0;
}
```

実行例

```
平均点を計算します。入力する
データ数を入れてください
5
10
20
30
40
50
入力されたデータ数に達しました
データ数=5,平均点=30
```