プログラミング及び演習
第7回 ポインタ (教科書第10章)
(2017/06/02)

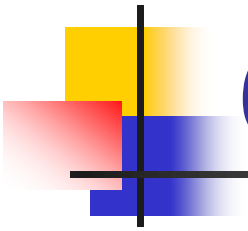
大学院情報学研究科知能システム学専攻
教授　森健策
大学院情報学研究科知能システム学専攻
助教　小田昌宏

本日の講義・演習の内容

- ポインタ
  - ポインタ　第10章
- 講義・演習ホームページ
  - http://www.newves.org/~mori/17Programming
- ところで，
  - さあ、いよいよポインタです。
  - コツさえわかれば難しくないので安心してください。

本日の講義資料について

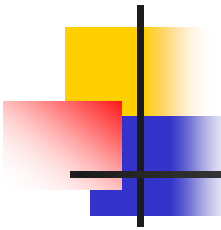
- 本日の講義資料は32bit環境で作成しているため、32bitアドレス空間でアドレスが表示されています。
- 64bit環境でコンパイルするとアドレスは64bitになります。
- 32bitだと4byteアドレスですが、64bitですと8byteアドレスになってしまうため、資料にアドレスを例示することが難しくなり32bitとしています。



64bit環境でコンパイルするには

- 64bitバイナリを生成するには
  - -m64 をつけてコンパイル
- 32bitバイナリを生成するには
  - -m32 をつけてコンパイル

```
ssh.ice.nuie.nagoya-u.ac.jp{mori}56: gcc -m32 test.c -o test
ssh.ice.nuie.nagoya-u.ac.jp{mori}57: file test
test: ELF 32-bit LSB executable, Intel 80386, version 1 (SYSV), dynamically linked
(uses shared libs), for GNU/Linux 2.6.18, not stripped
ssh.ice.nuie.nagoya-u.ac.jp{mori}58: gcc -m64 test.c -o test
ssh.ice.nuie.nagoya-u.ac.jp{mori}59: file test
test: ELF 64-bit LSB executable, x86-64, version 1 (SYSV), dynamically linked (uses
shared libs), for GNU/Linux 2.6.18, not stripped
```

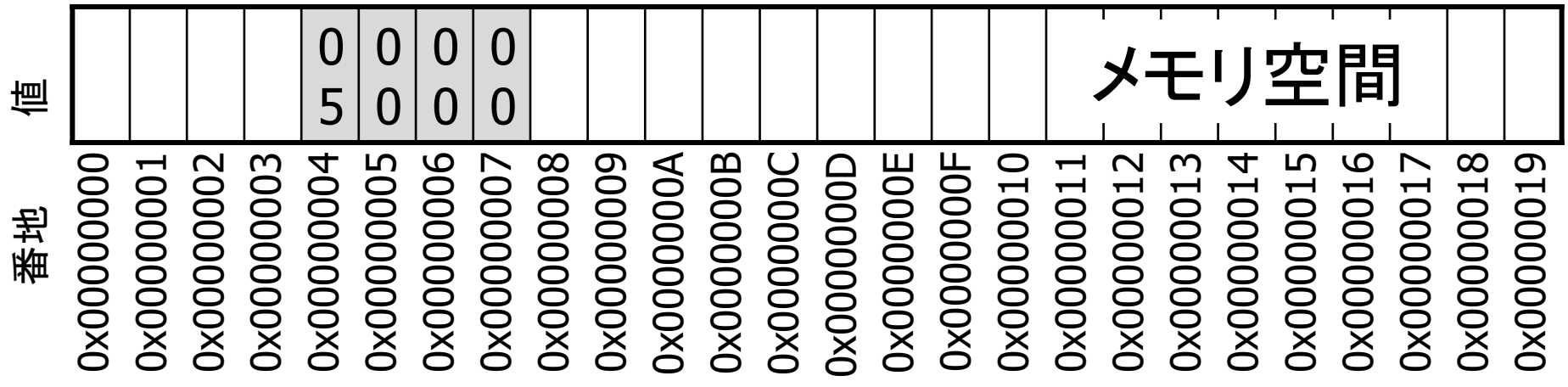


ポインタとは?

- 変数や関数などがメモリの中で格納されている場所
- メモリ空間上の「アドレス」を指し示す
- ポインタ変数
  - ポインタを取り扱うことのできる変数
- 一般的に
  - ポインタを使うプログラムは作成や理解が難しい
  - ポインタが常にどこを指すかを考える
  - ポインタ変数を初期化せずにそれ指し示す内容をアクセスするとセグメンテーション違反 (Segmentation Fault) が発生

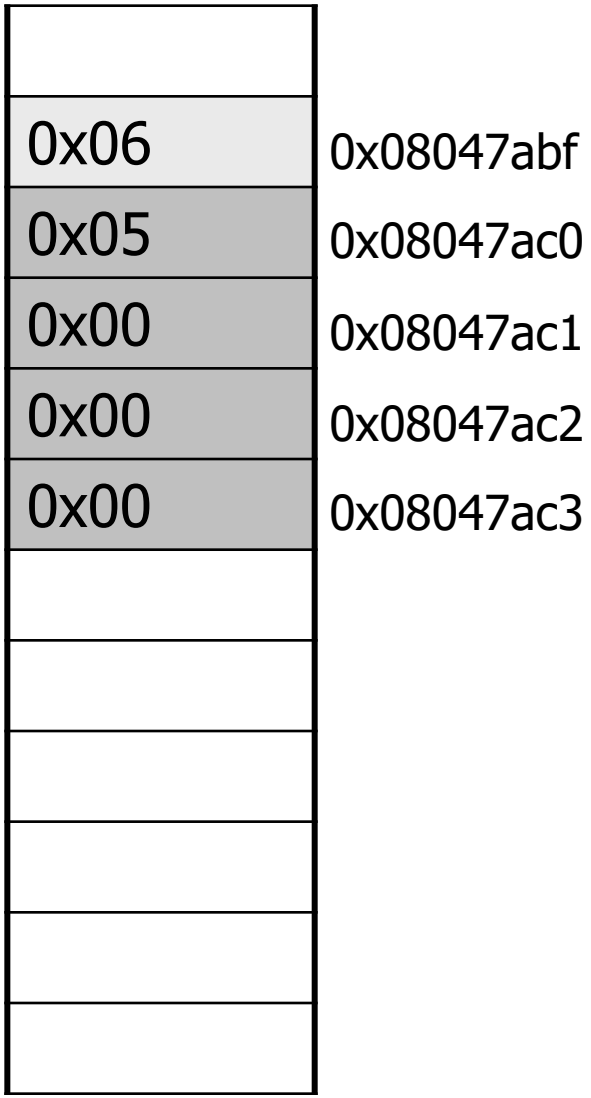
ポインタを理解するうえで

- 変数の内容はメモリ空間のどこかに記録される
- コンピュータのメモリ
  - 番地と値(大抵の場合1つの番地に1バイト)
  - 書き込み動作　"0x00001234番地に値0x26を書け"
  - 読み込み動作　"0x00001234番地の値を読め"
  - 複数バイトからなる変数は複数の番地を使って記録



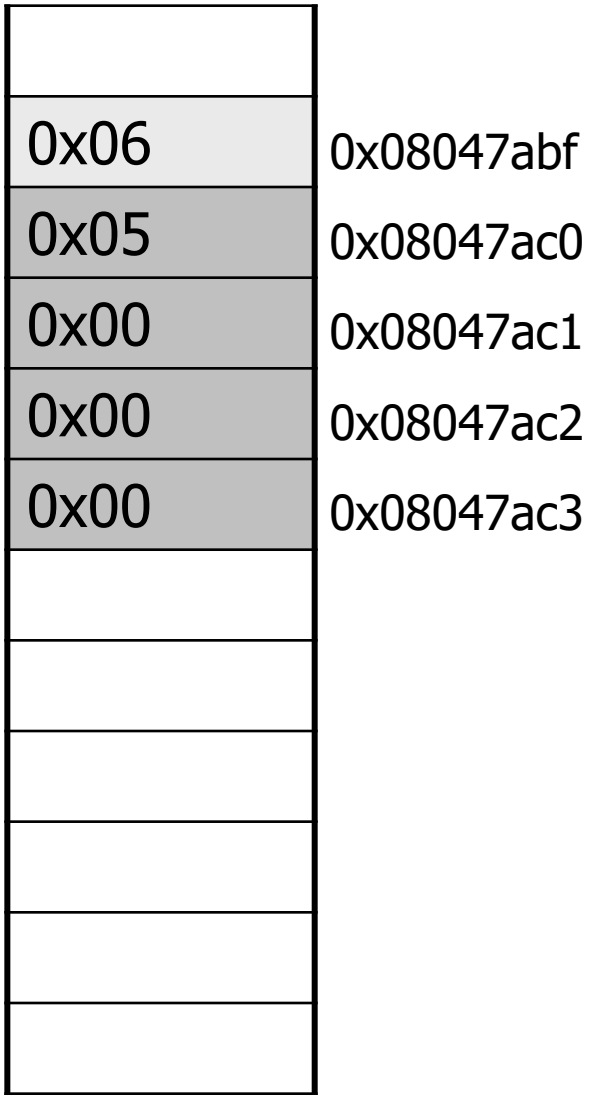
変数はどのように記憶されるか

- int i=5;とした場合
  - iの中身を記憶する場所が
    どこかに確保される
  - そこに0x00000005が
    書き込まれる (intなので4byte)
- char c=6;とすれば
  - cの中身を記憶する場所が
    どこかに確保される
  - そこに0x06が
    書き込まれる (charなので1byte)



ポインタとは

- メモリ空間での「番地」を表す
- それでは:
  - ある変数の中身が格納されている番地を知るにはどうするの?
  - 番地はわかっているのだけどそこに格納されている値は何?
- '*' と '&'が鍵となる記号
  - *pと書くとpが指し示すアドレス(番地)の内容を表す
  - &iと書くとiの中身が格納されているアドレス(番地)を表す

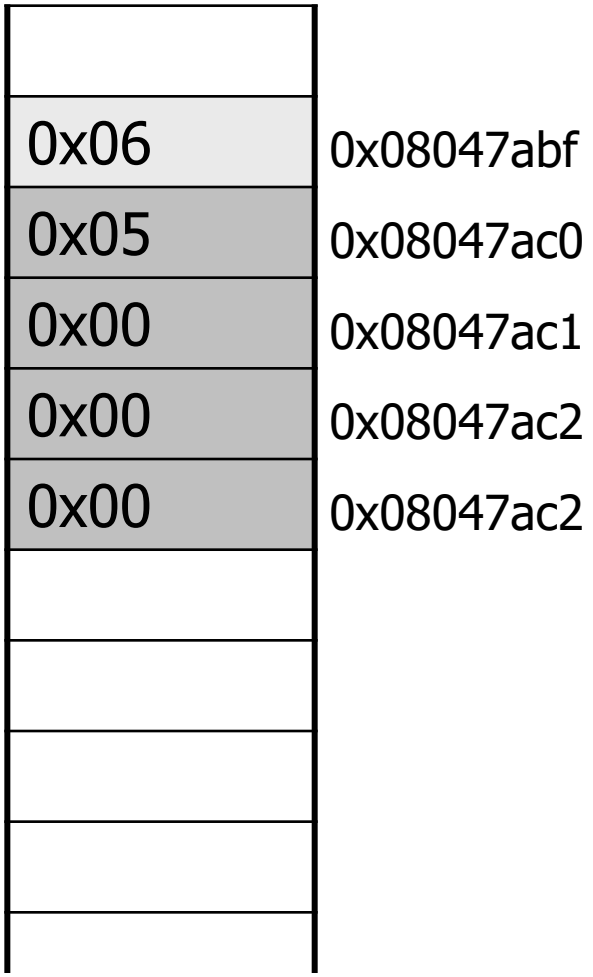


ポインタ変数の宣言

- '*'を変数名の前につける
  - int i, *p;　　pがint型データへのポインタ変数
  - int *p, q;　　pはポインタ変数　qは単なるint変数
  - int *p, *q;　pもqもポインタ変数
- 型はその番地にある値が表しているデータの型を示す
  - int *p; pが指し示す番地にはint型データがある
  - float *q; qが指し示す番地にはfloat型データがある
  - char *r; rが指し示す番地にはchar型データがある
  - 型によって何バイト分の値を使うかが異なる

番地を得るには?

- & 演算子を使う
- &iと記述すればiが格納される番地を表す
- 得られた番地はポインタ型変数に代入する
- 例
  - int i=5; char c=6;
    int *p1; char *p2;
    p1 = &i;
    p2 = &c;
  - p1には0x08047ac0
    p2には0x08047abf　が代入

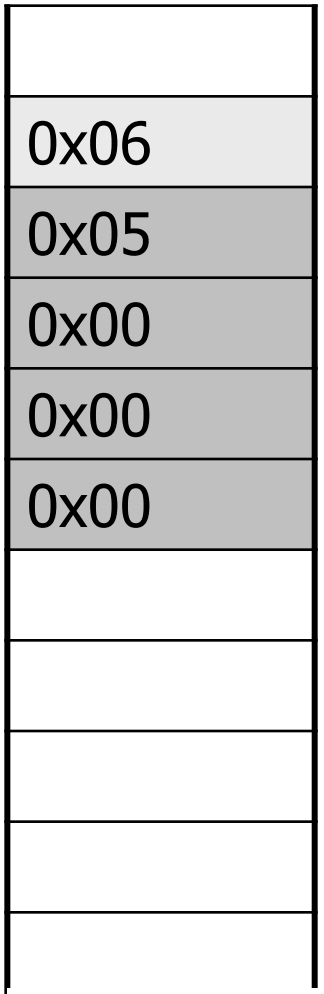


実際の番地は計算機毎に異なる

ここはあくまでも例!!

番地の内容を得るには?

- *演算子を使いその番地に書かれている内容を得る
- 注意: ポインタ変数の型で値を得る
  - charなら1byte分をアクセス
  - int なら4byte分をアクセス
  - int i=5; char c=6;
    int *p1; char *p2;
    p1 = &i;
    p2 = &c;
    printf("%d¥n",(*p1)+(*p2));
  - *p1は5
    *p2は6　を表す

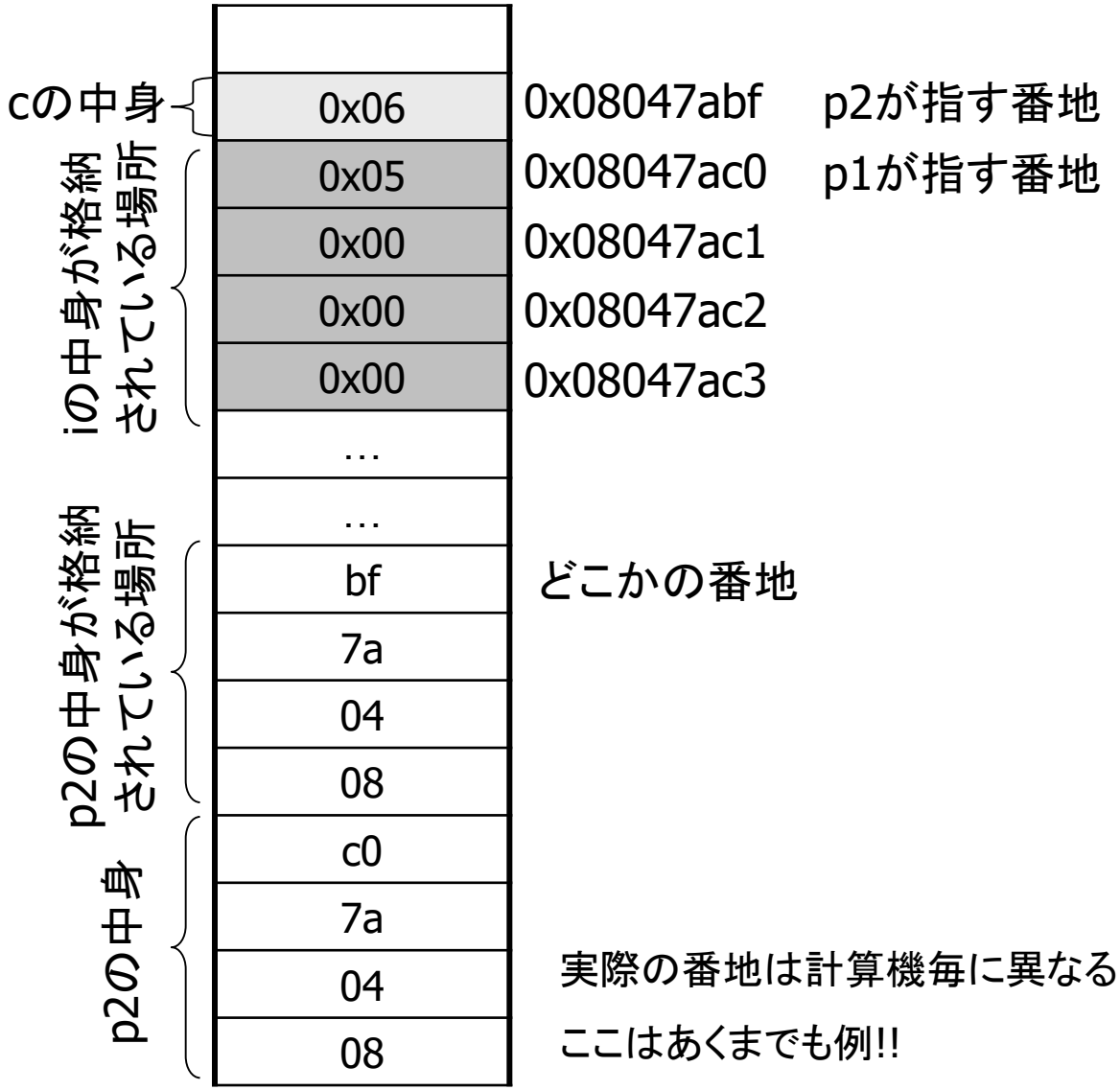


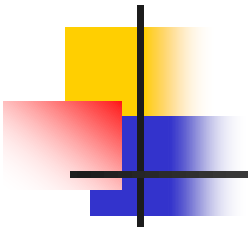
実際の番地は計算機毎に異なる

ここはあくまでも例!!

ポインタとメモリ空間

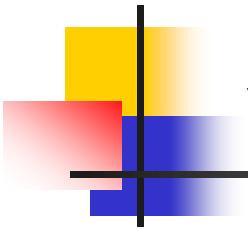
- int i=5;
  char c=6;
  int *p1;
  char *p2;
  p1=&i; p2=&c;





間接参照

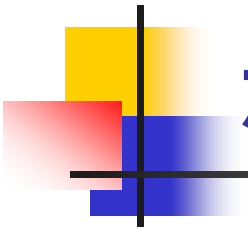
- ポインタを仲介して別の変数の値を読み書きすること
- 例
  - int i, j, *p, *q;
    p=q=&i;
    *p=1;
    p=&j
    *p=*q
- ポインタ変数自身に代入を行うこと，それが指し示している先に代入を行うことの違いに注意
- ポインタ変数が何も指していない状態
  - NULLポインタを用いる;
  - int *p;
    p = (int *) NULL;



やってはいけないこと

- ポインタ変数を初期化していないのにそれが指し示す内容を読み書きすること
  - そもそも初期化していないのだから，ポインタ変数がどこを指しているのか不定
  - 不定な場所を読み書きするため
    - 幸いな場合　→　Segmentation Faultが発生
    - 不幸な場合　→　プログラムは動くけど何か動作がおかしい
    - よくないプログラムの例

```
main()
{
  int *p;
  int a=6;
  *p = a;
}
```



ポインタを使う場合

- 関数引数での値の受け渡し
  - 参照呼出しを行う場合(Call by reference)
- 配列
  - 配列の関数での受け渡し
  - 配列の動的確保
- 構造体 (未学習)
  - 構造体の関数での受け渡し
  - 構造体の動的確保

関数側で引数を書き換える?

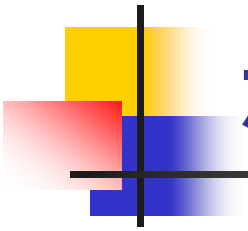
- 以下のプログラムを実行してもmain関数内のaの値はfunc内で書き換えられない
  - func関数呼び出し時にfuncのcに値がコピーされる
  - func関数実行後はcの中身は自動消去
-
```
void func(int c)
{
    c=c*5;
}
main()
{
    int a=5;
    func(a);
    printf("a=%d¥n",a);
}
```

関数側で引数を書き換える?

- 呼び出される側で呼び出す側の変数を書き換えるには参照渡し(ポインタ変数)を用いる
  - 呼び出す側はポインタを与える
  - 呼び出される側はポインタを使って呼び出した側の変数の中身をアクセス
- void func(int *c)
  {
  (*c)=(*c)*5;
  }
  main()
  {
  int a=5;
  func(&a);
  printf("a=%d¥n",a);
  }

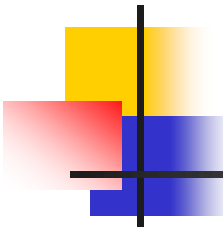
複数のポインタ変数を渡す

```
void func(int *c, int *d, int z)
{
    (*c)=(*c)*5+z;
    (*d)=(*d)*20-z;
}
main()
{
    int a=5, b=6, z=10;
    func(&a, &b, z);
    printf("a=%d b =%d¥n",a,b);
}
```



ポインタと配列

- ポインタと配列は類似した関係
  - 配列名は配列の先頭要素へのポインタ
  - ポインタの値を順次増加させるポインタが指し示す値への読み書き　→ 配列の要素番号を順次増加



ポインタと配列（例）

- int a[100]と書いたときaと&(a[0])は同じ
  - 配列名はポインタ変数
- int *p,a[10],b[5];
  p=a;
  p[1]=0; /* a[1]=0と同じ */
  p=b;
  p[1]=p[2]+3; /* b[1]=b[2]+3 と同じ */
  p=a;
  *(p+5) = *(p+1)+2; /* a[5]=a[1]+2 */

ポインタと配列

- int a[10];と宣言
- aは配列先頭へのポインタ
- *(a+2) で a[2]にアクセス可能
  - *(a+2) と a[2]は同じ
- (a+2)と記述したとき単に番地に2を足すのではなく(1要素のバイト数)*2を足した番地となる
  - 配列の2番目を正しくアクセス可能
  - 右の例だと
    a → 0x08047ac0
    a+1 → 0x08047ac4
    (int型=4byteのため)

変数を動的に割り当てる

- プログラムが開始されてから変数の記憶域を用意することが可能
  - (参考) これまではあらかじめ宣言
- 方法
  - 変数の記憶域を確保 (malloc関数)
  - 確保された場所はポインタ変数を使ってアクセス
  - 使い終わったら解放することを忘れずに (free関数)
    - 解放しないとメモリリークの原因

変数の動的割り当て例

- ```
  #include <stdlib.h>
  int *p;
  p = (int *)malloc(sizeof(int));
  *p = 1;
  …
  …
  free(p);
  ```
- malloc関数 (void *malloc(size_t size);)
  - 指定されたバイト数の領域を確保し，確保された領域へのポインタを返す
  - 確保できなかった場合にはNULLポインタを返す
  ```

配列を動的に割り当てる

- mallocで配列の要素分だけ記憶域を確保すればよい
- ```
  #include <stdlib.h>
  int *p, num;
  scanf("%d",&num);
  p = (int *)malloc(sizeof(int)*num);
  p[2] = 1;
  …
  …
  free(p);
  ```
- 任意の大きさの配列をプログラム実行中作成可能
  ```

関数に配列を渡す

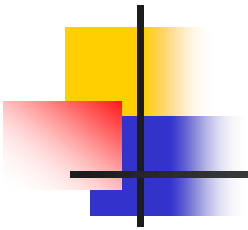
- 配列へのポインタを渡せばよい
-
```
void func(int *c, int num)
{
	int i;
	for(i=0;i<num;i++){
		c[i] = i;
	}
}
main()
{
	int a[5];
	func(a,5);
}
```

-
```
void func(int c[], int num)
{
	int i;
	for(i=0;i<num;i++){
		c[i] = i;
	}
}
main()
{
	int a[5];
	func(a,5);
}
```

どちらも内容的には同じ

関数に配列を渡す

```
void func(int *c, int num)
{
        int i;
        for(i=0;i<num;i++){
                c[i] = i;
        }
}
main()
{
        int *a,num=5,i;
        a = (int *)malloc(sizeof(int)*num);
        func(a,num);
         for(i=0;i<num;i++){
                printf( "a[%d] = %d¥n",i,a[i]);
        }
        free(a);
}
```



関数ポインタ

- 関数のエントリポイントのアドレスを保持する変数
- プログラムをコンパイルすると
  - プログラムの各関数についてエントリポイントを作成
  - 関数が呼び出されると実行制御はエントリポイントに移行
  - エントリポイント＝アドレス
- 関数へのポインタが得られればポインタを使って関数を呼び出すことが可能

関数ポインタの作成

- ポインタをその関数の戻り値の型と同じ型を持つポインタ変数と宣言
- 仮引数があれば続けて宣言
- 宣言の例
  - int (*p)(int x, int y); /* *pは括弧でくくる */
- 関数ポインタ取得例
  - int sum(int a, int b);という関数がある場合
  - p = sum; とすれば関数ポインタ取得可能
  - 呼び出しは result = (*p)(d,e); のように行う
  - result=p(d,e);でもOK

関数ポインタと呼び出し例

```
#include <stdio.h>
int add(int x, int y)
{
  return(x+y);
}
int mult(int x, int y)
{
  return(x*y);
}
int main()
{
  int(*fp)(int,int);
  fp=add;
  printf("add %d¥n",(*fp)(2,3));
  fp=mult;
  printf("mul %d¥n",(*fp)(2,3));
}
```

関数に関数ポインタを渡す

- 関数の引数として関数を渡すことが可能
  - 関数ポインタを利用する
  - ある関数の機能の一部を変化させる目的で使用される
- 代表的な使用例
  - C標準関数のqsort関数
  - 大小比較関数は関数ポインタとしてqsort関数に渡す必要あり
  - 引数として渡す大小比較関数を変えることでさまざまなデータのソーティングが可能

関数に関数ポインタを渡す例

```
#include <stdio.h>
int add(int x, int y)
{
  return(x+y);
}
int mult(int x, int y)
{
  return(x*y);
}
```

```
int
compute(int(*fp)(int,int),int v, int array[], int size)
{
  int w,i;
  for(w=v,i=0;i<size;i++){
    w=(*fp)(w,array[i]);
  }
  return w;
}
int main()
{
  int nums[10]={1,2,3,4,5,6,7,8,9,10};
  printf("addsum %d¥n", compute(add,0,nums,10));
  printf("mulsum %d¥n", compute(mult,1,nums,10));
}
```

qsort関数

- 比較関数を関数ポインタを用いて関数に渡す
- void qsort(void *base, size_t nel, size_t width, int(*compar)(const void *, const void *));

```
#include <stdio.h>
#include <stdlib.h>
int intcompare(const void *p1,
               const void *p2)
{
  int i = *((int *)p1);
  int j = *((int *)p2);
  if (i > j)
    return (1);
  if (i < j)
    return (-1);
  return (0);
}
```

```
int main()
{
  int i;
  int a[10] = { 9, 8, 7, 6, 5, 4, 3, 2, 1, 0 };
  size_t nelems = 10;
  qsort((void *)a, nelems, sizeof (int),
intcompare);
  for (i = 0; i < nelems; i++) {
    (void) printf("%d ", a[i]);
  }
  (void) printf("¥n");
  return (0);
}
```

qsortのman page

名前

qsort - 配列を並べ変える

書式

```
#include <stdlib.h>
void qsort(void *base, size_t nmemb, size_t size,
        int(*compar)(const void *, const void *));
```

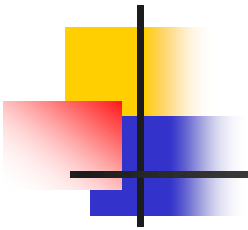
説明

qsort() 関数は、大きさ size の nmemb 個の要素をもつ配列を並べ変える。base 引数は配列の先頭へのポインタである。compar をポインタとする比較関数によって、配列の中身は昇順 (値の大きいものほど後に並ぶ順番) に並べられる。比較関数の引数は比較されるふたつのオブジェクトのポインタである。

比較関数は、第一引数が第二引数に対して、1) 小さい、2) 等しい、3) 大きいのそれぞれに応じて、1) ゼロより小さい整数、2) ゼロ、3) ゼロより大きい整数のいずれかを返さなければならない。ふたつの要素の比較結果が等しいとき、並べ変えた後の配列では、ふたつの順序は定義されていない。

返り値

qsort() は値を返さない。

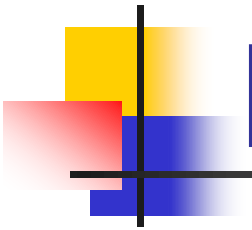


関数ポインタの配列

- 関数ポインタを配列に格納したもの
  - 配列の各要素は異なる関数を示す
    (要素として関数へのエントリポイントが格納される)
- 添え字を変更するだけで色々な関数が呼び出し可
  - 状況に応じて呼び出す関数を簡単に変更できる
- 宣言例
  - int (*p[4])(int x, int y);
  - int (*ops[])(int x, int y)={add, mult};

関数ポインタ配列の例

```
#include <stdio.h>
int add(int x, int y)
{
  return(x+y);
}
int mult(int x, int y)
{
  return(x*y);
}
int main()
{
  int (*ops[6])(int x, int y);
  int i;
  ops[0] = ops[2] = ops[4] = add;
  ops[1] = ops[3] = ops[5] = mult;
  for(i=0;i<6;i++){
    printf("ops %d¥n",(*ops[i])(2,3));
  }
}
```



関数ポインタ配列の例

```
#include <stdio.h>

int add(int x, int y)
{
  return(x+y);
}
int mult(int x, int y)
{
  return(x*y);
}
int main()
{
  int (*ops[])(int x, int y)={add,mult,add,mult,add,mult};
  int i;
  for(i=0;i<6;i++){
    printf("ops %d¥n",(*ops[i])(2,3));
  }
}
```