システムプログラミング序論
第4回
ポインタ

大山恵弘

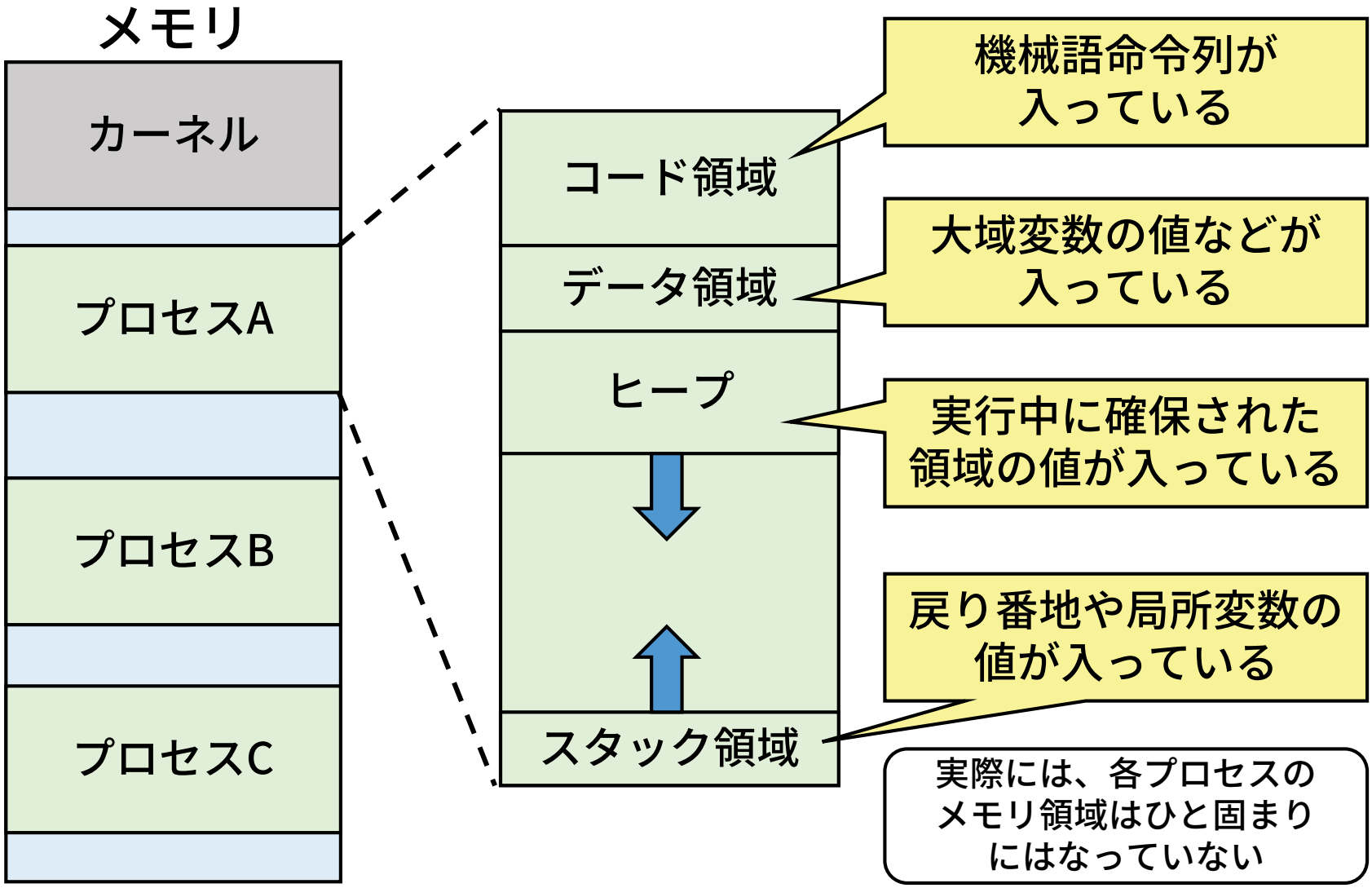
ポインタ

- C言語の特徴の1つ
  - 便利でもあり，わかりにくくもあり，危険でもあり，...
- 計算機の仕組みと密接に関係
  - 計算機の生の姿をプログラマに見せてくれる
  - 低い（ハードウェアに近い）レベルのプログラム（OSやデバイスドライバ）でC言語がよく使われる理由となっている

ポインタを一言で言うと

- ポインタとはアドレスである！
- では，アドレスとは？
  → CPUがメモリをアクセスするときに指定する，
  メモリ上の場所
  - メモリはデータを入れる箱が並んだものであり，
    アドレスは箱に付けられた番号
  - メモリは1バイトごとに区切られているので，
    何バイト目をアクセスするかを指定
    - たとえば32 bit環境で2GBのメモリなら0x00000000～0x7fffffff
      のどこなのか（0～2147483647のどこなのか）を指定

プロセスのメモリイメージ



プロセスのアドレス空間

- Linux上で `cat /proc/プロセス番号/maps` を実行してみよう

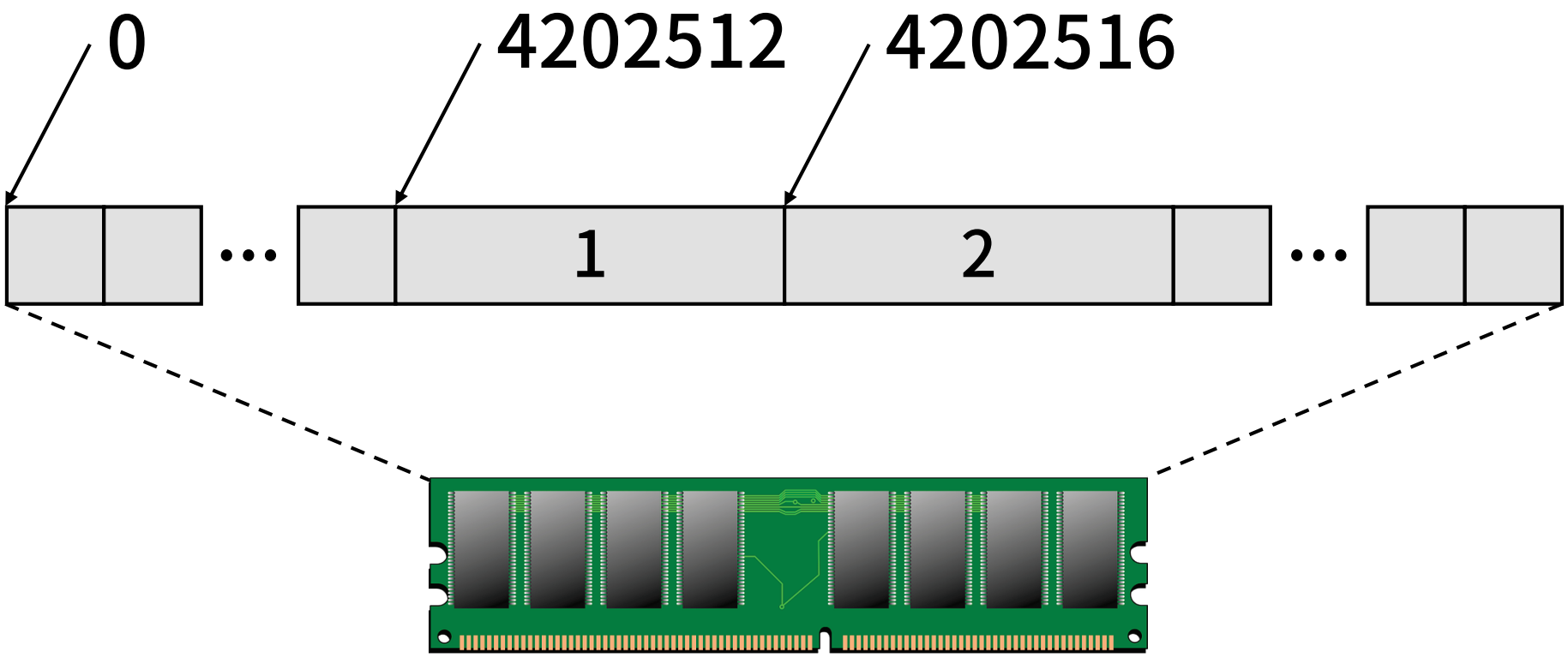
アドレスを得るには？

- 変数に & を付けるとその変数へのポインタが得られる
  - ハードウェアのレベルで言うならば，その変数が格納されたメモリ上の場所を示すアドレスが得られる
  - 「ポインタが指す変数」や「ポインタが指すデータ」という表現がよく用いられる

```
int x = 1;
int y = 2;
printf("x = %d, y = %d¥n", x, y);
printf("&x = %d, &y = %d¥n", &x, &y);
```

```
x = 1, y = 2
&x = 4202512, &y = 4202516
```

イメージ



ポインタ変数

- ポインタ（アドレス）を格納するための変数
- ポインタ変数の宣言では，そのポインタの場所にあるデータの型を指定しなくてはならない
  - そのポインタは `int` 型のデータの場所を示しているのか，`double` 型のデータの場所を示しているのか，など
- 宣言の例

```
int *p;
```

- 宣言の書き方： そのポインタの場所にあるデータの型を書き，変数名の前に `*` を付ける

ポインタ変数の使い方

- ポインタ変数へのポインタの代入には = を使う

```
p = &a; /* a を指すポインタの値を p に代入する */
```

- ポインタが指すデータを得るには * を使う

```
x = *p; /* p が指すデータを x に代入する */
```

- ポインタが指す場所にデータを格納するにも * を使う

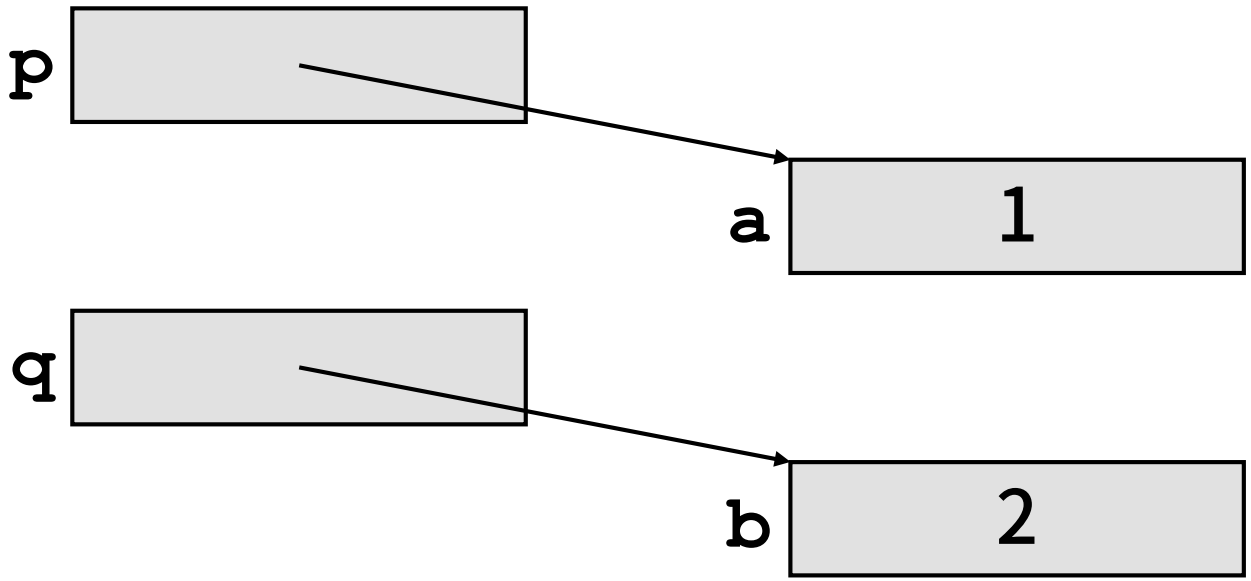
```
*p = 100; /* p が指す場所に100を代入する */
```

- ポインタの宣言にも，ポインタを用いたデータの獲得にも，格納にも，同じ記号 * が用いられることに注意
  - 学習者が混乱してつまづく1つの理由

何が表示されるでしょう？

```
int a, b;
int *p, *q;
a = 1;
b = 2;
p = &a;
q = &b;
*p = *p + 1;
*q = *q + *p;
printf("a=%d, b=%d¥n", a, b);
```

イメージ



関数とポインタ

- ポインタを引数として受け取る関数を定義するには，関数の引数をポインタ変数とする
  - 例：2つの変数の値を交換する関数 **swap**
    - ポインタがないと書けない！

```
void swap(int *p, int *q)
{
  int t;
  t = *p;
  *p = *q;
  *q = t;
}
```

```
int main(void)
{
  int a, b;
  a = 100;
  b = 2;
  swap(&a, &b);
  printf("a=%d¥n", a);
  printf("b=%d¥n", b);
  return 0;
}
```

これと比較せよ

```
void swap(int p, int q)
{
  int t;
  t = p;
  p = q;
  q = t;
}
```

```
int main(void)
{
  int a, b;
  a = 100;
  b = 2;
  swap(a, b);
  printf("a=%d¥n", a);
  printf("b=%d¥n", b);
  return 0;
}
```

関数から複数の値を返したいとき

- これまでの説明では，関数が返り値として返せる値は１つだけだった
- ポインタの引数を使えば複数の値を「返せる」
  - 例： 足し算と引き算の値を両方返す関数

```
void addsub(int x, int y, int *sum, int *diff)
{
  *sum = x + y;
  *diff = x - y;
}
```

配列とポインタ

- 配列とポインタは別のもの
  - 配列はデータの集まり
  - ポインタはデータを指すためのデータ
- しかし，混同しやすい
  - `int a[100];` で配列 `a` を宣言したとする
  - `a[n]` は配列 `a` の `n` 番目の要素の整数を表す
  - では，（単独で使う） `a` は，型は何で，値は何？

```
int a[100];
printf("%d¥n", a);
```

配列名のデータの型と値

- 配列が置かれたメモリの先頭アドレスを表す
- すなわち，ポインタを表す
- よって，ポインタ変数に代入することもできる
- printf でポインタの値（アドレス）の表示もできる

```
int a[100];
int *b;
b = a;
/* 表示されるのは，ともに，a の先頭アドレス */
printf("%d %d¥n", a, b);
```

ポインタの演算

- ポインタに整数を足すことができる
  - ポインタ p に1を足した結果は，p が指すデータの一つ次の要素を指すポインタ
  - 同様に，n を足した結果は，n 個後の要素を指すポインタ
  - ポインタの足し算において，ポインタの値の変化幅は，ポインタが指すデータの型に依存する
    - 「一つ次の要素」が何バイト先にあるかに依存する
      - ポインタが64bit整数を指しているなら8バイト先
      - ポインタが文字を指しているなら1バイト先

```
int a[100];
int *b;
b = a;
b = b + 1;
/* 表示される値の差は，多くの処理系では4 （1ではない） */
printf("%d %d¥n", a, b);
```

ポインタとカッコ [] の組み合わせ

- p がポインタ，n が整数であるとき，p[n] は，ポインタ p に n を足した場所にあるデータを表す
  - つまり，*(p + n) を表す
  - p[n] と *(p + n) は同じもの

```
int a[100];
int *b;
a[2] = 9;
b = a;
/* 表示されるのは，ともに，9 */
printf("%d %d¥n", a[2], b[2]);
```

余談

- すべて同じアドレスから値を読み込んで返す
  - a[10]
  - *(a + 10)
  - *(10 + a)
  - 10[a]

ポインタの引き算

- ポインタから整数を引くこともできる
  - ポインタ p から n を引いた結果は，p が指すデータの n 個前の要素を指すポインタ
- ポインタの差を得ることもできる
  - 差とは「何要素分離れているか」
  - ポインタ q からポインタ p を引いて得られる差 d とは，p + d = q となる値

```
int *p, *q;
p = &a[2];
q = &a[10];
d = q - p; /* 8要素分離れているのでdは 8 */
printf("%d %d¥n", p, q); /* 表示される値の差は8ではない */
```

scanf, again

- scanf の引数には，入力された値を入れる場所（アドレス）を与えることになっている
- だからこうなる

```
int x;
char s[256];
scanf("%d", &x); /* x の値ではなく x の場所を渡す */
scanf("%s", s); /* s の中身ではなく s の場所を渡す */
```

関数引数に出現する配列とポインタ

- 関数に配列を渡そうとしても，実際に渡るのは（配列の先頭要素への）ポインタである
- だから左の書き方が実状に近いが，右の書き方も使える（左と右は完全に等価）

```
void foo(int *a)
{
   ...
   x = a[3]; /* OK */
   y = *(a + 5); /* OK */
   ...
}
```

```
void foo(int a[])
{
   ...
   x = a[3]; /* OK */
   y = *(a + 5); /* OK */
   ...
}
```

細かいことを言えば，「関数引数における配列では，1つ目の次元の要素数は無視され，配列はポインタに置き換えられる」と仕様で決まっている

`int main(int argc, char *argv[])` と
`int main(int argc, char **argv)` も等価

データ型

- 変数や値がどのような種類のデータであるか
- 基本データ型
  - `int`
  - `double`
  - `char`
  - …
- 配列データ型（配列型）
  - `int a[100]; /* データ型そのものの記述ではないが */`
  - `int matrix[3][3]; /* 上に同じ */`
- ポインタデータ型（ポインタ型）
  - `char *`
- …

おさらい

- ポインタはアドレスであり，メモリ上の位置である
- ポインタが指すアドレスに対して，値の読み出しや書き込みができる
- ポインタを変数に入れることもできる
- ポインタを関数の引数として与えることもできる
- 配列の先頭や中間を指すポインタを作れる
- `p` がポインタ型の変数であるとき，`p + 1` は `p` の値（アドレス）に1を足した値であるとは限らない

変数の有効範囲

変数の有効範囲と有効期限

- 有効範囲（scope）
  - どこで使えるか
  - 局所変数： 変数が宣言/定義された関数内の，その宣言/定義以降の部分
  - 大域変数： その変数の宣言/定義以降の部分
- 有効期限（extent）
  - データがいつまで有効か（保持されるか）

変数の有効期限（extent）

- 局所変数の値（配列の内容も含まれる）は，その変数が定義された関数が終了すると無効になる
  - その変数のメモリが解放される

```
char *buggy(void)
{
    char buf[4096];
    ...
    return buf;
}
```

- 大域変数の値はプログラムの実行全体を通じて保持される

静的変数（static variable）

- 局所変数に `static` をつける
- 有効範囲は局所変数と同じ
- 有効期限は大域変数と同じ
- 参照や更新をする場所は限られるが，値を長く保持すべき変数を導入したい場合によく使う
- 関数にも `static` がつけられる

```
int get_next_prime(void)
{
  static int candidate = 2;
  while (!is_prime(candidate)) {
    candidate++;
  }
  return candidate;
}
```

```
get_next_prime() → 2
get_next_prime() → 3
get_next_prime() → 5
...
```

static 付きで定義された関数

- そのファイル内でしか使えない
  - そのファイルの外からは，その関数を呼び出せない

```
int plus1(int x)
{
  return x + 1;
}

static int minus1(int x)
{
  return x - 1;
}
```

sub.c

```
#include <stdio.h>

int main(void)
{
  a = 10;
  printf("%d¥n", plus1(a));
  printf("%d¥n", minus1(a));
  return 0;
}
```

OK

Error

main.c

乱数

乱数を生成するための ライブラリ関数

- int rand(void);
  - 0以上 RAND_MAX－1 以下の範囲からランダムに選んだ整数を返す
    - いわゆる疑似乱数
- void srand(unsigned int seed);
  - rand 関数が返す乱数列のパターンを seed と指定する
    - いわゆる疑似乱数列の種
    - 同じ種を使うと同じ乱数列が生成される
- おそらく #include <stdlib.h> を最初に書く必要がある
- 使用例：
  - `int a = rand() % 10;`
    - `a` に0以上9以下の乱数を代入
  - `double f = (double)rand() / (double)(RAND_MAX);`
    - `f` に [0, 1.0) の範囲の小数（浮動小数点数）を代入