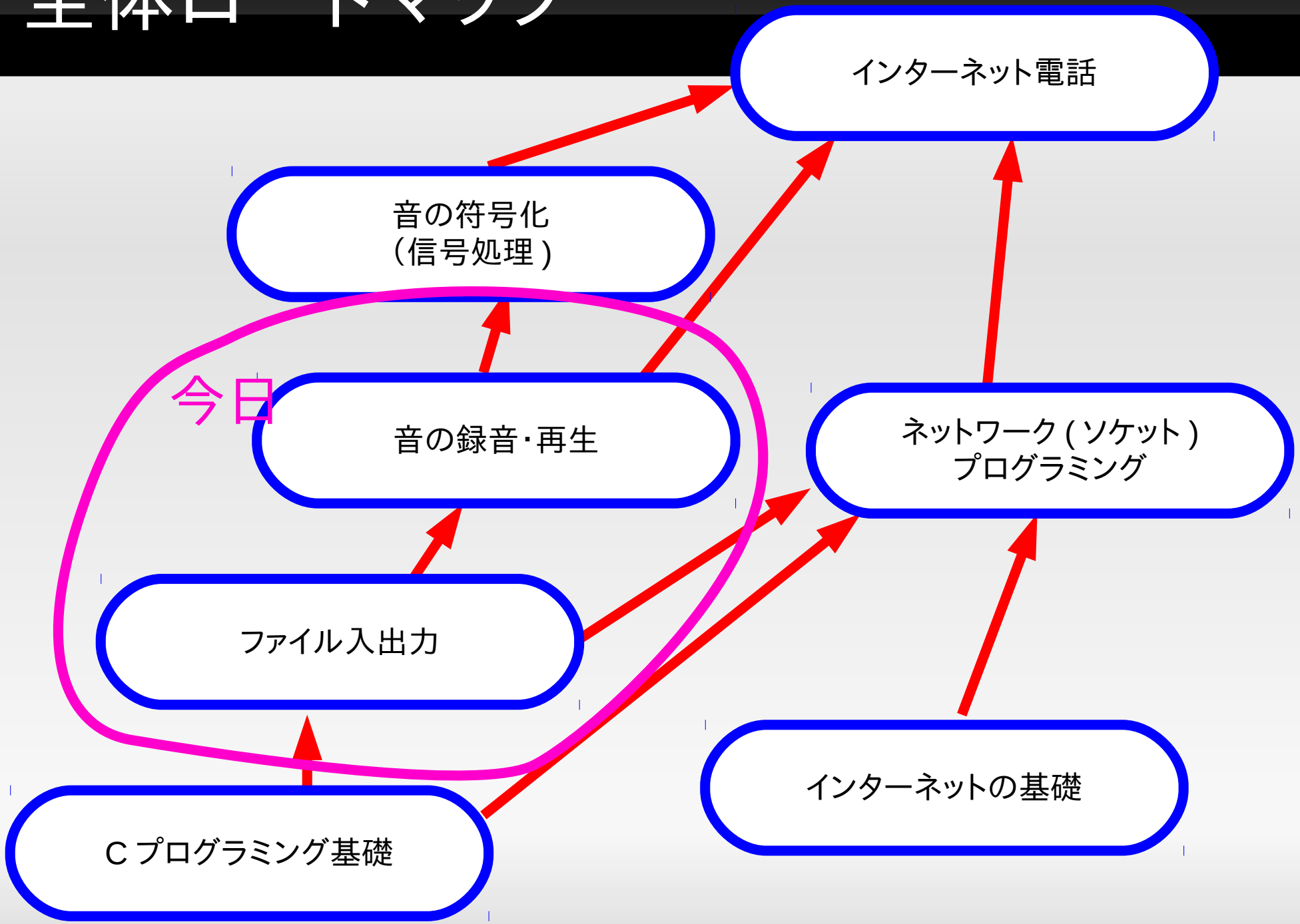
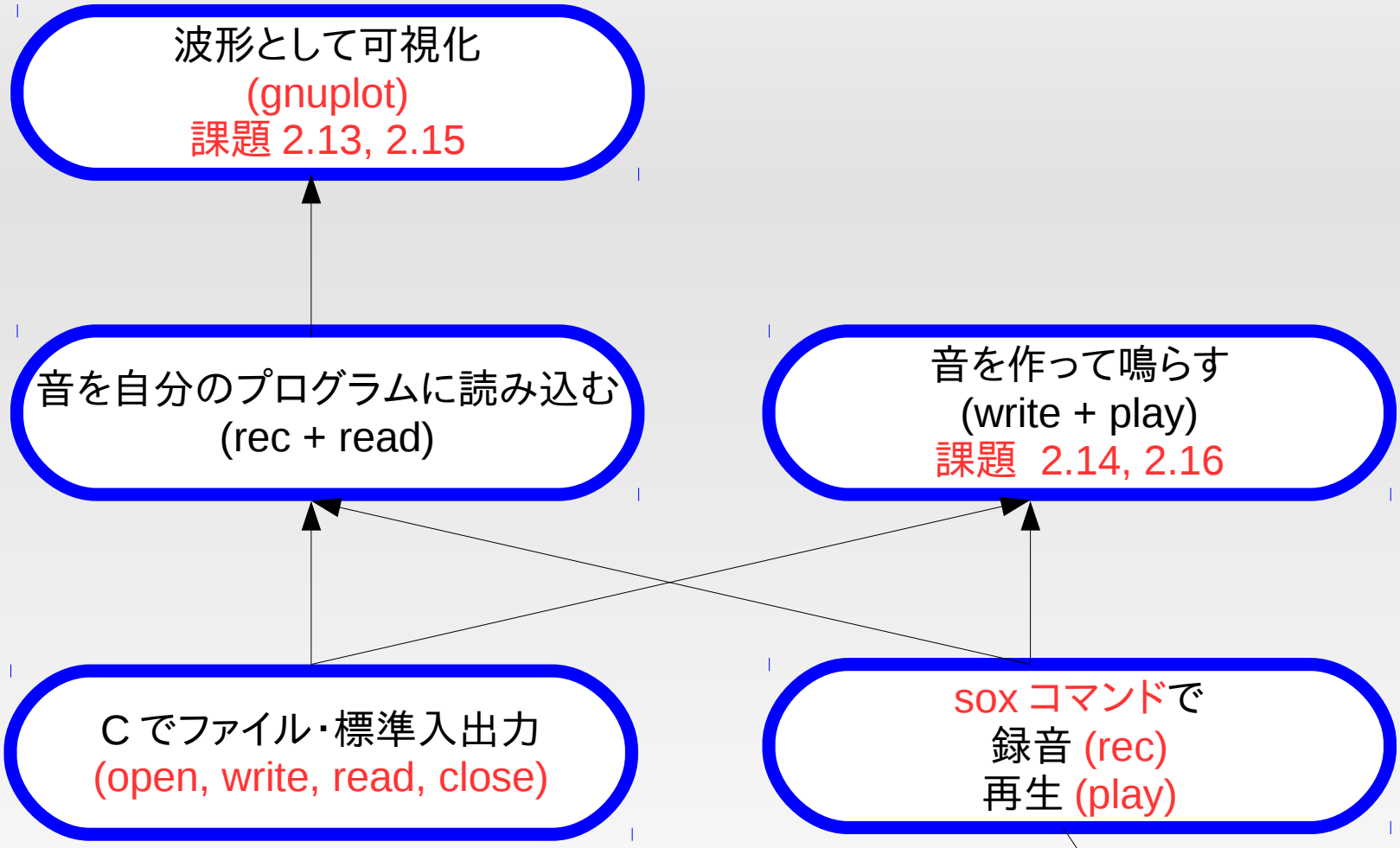
ファイル入出力

全体ロードマップ


インターネット電話
音の符号化
(信号処理)
今日
音の録音・再生
ネットワーク(ソケット)
プログラミング
ファイル入出力
インターネットの基礎
Cプログラミング基礎

今日のロードマップ



音がデータの列としてどう表されているのか (符号化) を理解
特に，Linear PCM (raw 形式). 標本化周波数 , 量子化 bit 数 , チャネル数

ファイル入出力の流れ

- 書き込み , 作成
  - open; write (任意回); close
- 読み込み
  - open; read (任意回); close
- man -s 2 open （または read, write, close) で必要な #include, 引数の意味などを調べてみよ

書き込み・作成

```
int fd = open(filename, O_WRONLY|O_CREAT|O_TRUNC, 0644);
```

確かに *filename* を開いたよという「印」(切符)
ファイルディスクリプタ． 実体はただの整数 (3, 4, 5, ...)

```
m = write(fd, data, n);
```

書きたいバイト数
配列 (ポインタ;アドレス)

```
close(fd);
```

実際に書けたバイト数
もしくはエラー発生時は -1

新たな超重要注意：「右を見て左を見て，また右を見て…」

- システム関数の呼び出しは「失敗するもの」と思って書く
  - 呼び出したら成功を確認してから先へ進む
- 絶対駄目：

```
int fd = open(filename, ...);
write(fd, ..., ...);
```

- ないよりマシ：

```
int fd = open(filename, ...);
if (fd == -1) { printf(”gaan\n”); exit(1); }
```

- 推奨：

```
int fd = open(filename, ...);
if (fd == -1) { perror(”open”); exit(1); }
```

おすすめスタイル

- 一度だけ書いておく；こんだけ！

```
void die(char * s) { perror(s); exit(1); }
```

- 何かあったらすぐ die

```
NG: int fd = open(...);
```

```
OK: int fd = open(...);
    if (fd == -1) die(”open”);
```

- ” エラー時には errno をセットする”関数 (man を見よ) は，エラー直後に perror を呼べば有用情報が表示される

読み込み

```
int fd = open(filename, O_RDONLY);
```

```
m = read(fd, data, n);
```

書きたいバイト数

n バイト以上ある配列 (ポインタ ; アドレス)

```
close(fd);
```

実際に書けたバイト数 (エラー時は -1)

「何を書いているか」誤解なきよう
以下の違い・同じが区別できるように

- char a[4] = { 1, 2, 3, 4 };
  write(fd, a, 4);
- char a[4] = { '1', '2', '3', '4' };
  write(fd, a, 4);
- char * a = ”1234”;
  write(fd, a, 4);
- int a[4] = { 1, 2, 3, 4 };
  write(fd, a, 4);
- fprintf(fp, ”%d”, 1234);

概念整理 (ファイルの中身)

- コンピュータは全てを 0/1 (bit) で表すんだって !
- 通常 , 最低でも 8 つの bit (普通これを 1 byte と呼ぶ) を一まとめにして扱う (16, 32, 64 bit などの場合もある)
- いいかえればすべてのファイルは byte (256 種類のデータ ; 0-255 のどれか) がずらーっと並んだ物
  - -128 – 127 のどれかと思ったりする場合もあり , 2 バイトずつまとめて 0-65535 の列と思ったり , 都合に応じて「解釈」は変わる

混乱したら数字しか世の中にないと思うが吉

- あえて標語的に言えば
  - 文字列 , 文字などというものは存在しない
  - 'a' → じつは 49 のこと (ascii 符号)
  - ”abc” → じつは { 49, 50, 51, 0 } のこと
- 「数字データ」(または「バイト列」) しか世の中にはなく , それ以外のものはその「バイト列」の解釈方法 (「符号化」) によって作られている幻想？に過ぎないと思っておけば良い

以下の違い・同じが区別できるように

- char a[4] = { 1, 2, 3, 4 };
  write(fd, a, 4);
- char a[4] = { '1', '2', '3', '4' };
  write(fd, a, 4);
  → 実は char a[4] = { 49, 50, 51, 52 }; と同じ
- char * a = ”1234”;
  write(fd, a, 4); → 上と同じ
- int a[4] = { 1, 2, 3, 4 };
  write(fd, a, 4);
  char a[16] = {1,0,0,0,2,0,0,0,3,0,0,0,4,0,0,0} と同じ
  実際に書かれるのは , {1,0,0,0} まで (あくまで 4 バイト)
- fprintf(fp, ”%d”, 1234);
  2,3 番目の例と同じ (fprintf の中でややこしい変換している)

od (octet dump) コマンド

- ファイル中の「バイト列」を読める数字の列で表示してくれるコマンド
- 基本 : od -t u1 ファイル名 でファイルの各バイトを 0 … 255 で表示する
  - char a[4] = { 1, 2, 3, 4 }; → 1 2 3 4
  - char a[4] = {'1','2','3','4'}; → 49 50 51 52
  - ”1234” → 49 50 51 52
  - int a[4] = { 1, 2, 3, 4 } → 1 0 0 0 2 0 0 0
- オプションしだいで 2 バイト一組 , 4 バイト一組 , …, 符号あり･なしなどでの表示も可能

補足 : fopen, fwrite, fread, fclose

- open; write/read; close の代わりに , fopen, fwrite/fread, fclose という関数もある

fopen を用いたファイル作成

```
FILE * fp = fopen(filename, ”w”);
```

確かに *filename* を開いたよという「印」(切符)
ファイル構造体

```
m = fwrite(data, s, n, fp);
```

書きたいバイト数 (要素サイズ s x 要素数 n)
配列 (ポインタ;アドレス)

```
fclose(fp);
```

実際に書けた要素数

両者の違い

- Unix においては , open/read/write が , ”the” プリミティブ (OS のシステムコール)
  - fopen は open, fread は read, … を使っているだけ
- ユーザから見た違い
  - 多くの目的は当然どちらでも達成できる . 「混ぜるなキケン」とだけ覚えておけば良い
  - fopen 系には気の利いた機能もある
  - fgets (改行まで読む), fprintf (書式付き出力), fscanf
  - fopen を使いたくない理由は「バッファリング」

バッファリング

- fwrite write だが ,
  - write : その場で即 OS に「書け」
  - fwrite : 少しデータがたまったところで一括して write
- 普段はありがたい機能 (write を呼ぶオーバーヘッドを低減)
- 一方 fwrite で「書いたつもりなのにデータがファイルに反映されない , 音がすぐにならない」などの問題はバッファリングが原因になることもある

標準入出力，リダイレクト，パイプ

- 標準入出力
  - open しなくても「最初からある」ファイルディスクリプタ
- リダイレクト
  - 自分で open しなくても，シェルがファイルを開いて標準入出力にしてくれる
- パイプ
  - 自分で open しなくてもシェルが，自分の標準入 (出) 力と，他のプロセスの標準出 (入) 力を結んでくれる

標準入出力

- ファイルディスクリプタ 0, 1, 2 のこと
  - 0： 標準入力
  - 1： 標準出力
  - 2： 標準エラー出力
- つまり以下は , open もせずにいきなりやってよい
  - m = read(0, data, n);
  - m = write(1, data, n);
  - m = write(2, data, n);

それぞれ何なのか？

- 普通は，
  - 標準入力 ： 端末からのキーボード入力
  - 標準出力 ： 端末への出力
  - 標準エラー出力 ： 端末への出力
- つまり，
  - read(0, data, n) → キーボードから読む
  - write(1, data, n) → 端末へ書く
  - write(2, data, n) → 端末へ書く

リダイレクト

- シェルの機能

```
$ コマンドライン > filename
```

- と書くだけで「コマンドライン」の標準出力を *filename* にしてくれる
- 標準入力

```
$ コマンドライン < filename
```

- 標準エラー出力 (あまり使わない ; エラーメッセージを保存したい時とか)

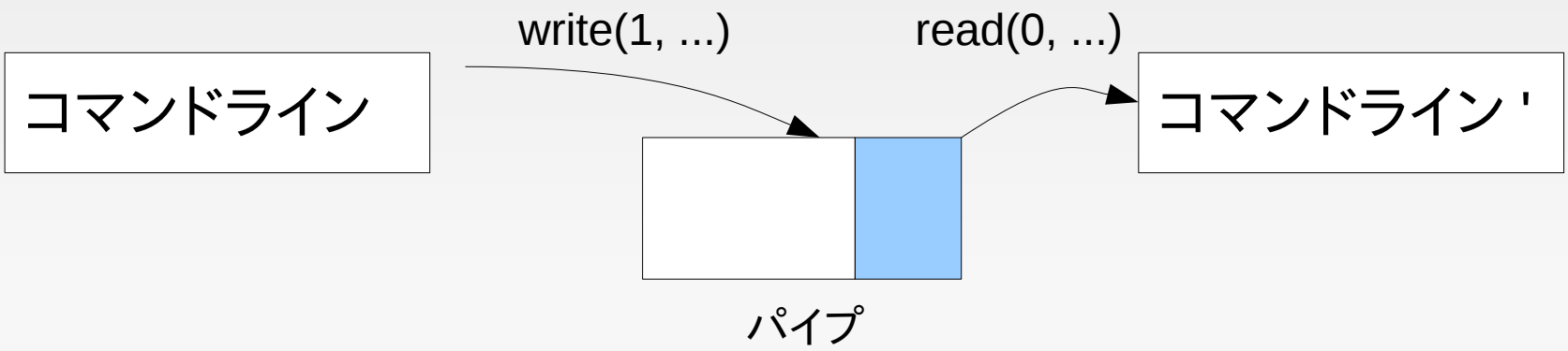
```
$ コマンドライン 2> filename
```

パイプ

- シェルの機能

```
$ コマンドライン | コマンドライン '
```

- とかくだけで ,
  - 「コマンドライン」の標準出力を「コマンドライン ' 」の標準入力へつなげてくれる



どれも Unix の地味だが偉大な発明

- ファイルディスクリプタの概念
  - 入出力先がなんであっても (ファイル , キーボード , 端末 , パイプ , ネットワーク), write/read を使えば良い
- リダイレクト
  - 必要に応じてあちらに書いたりこちらに書いたり , というアプリが簡単に書ける
  - 標準入出力を使えばその「あちらやこちら」を自分で書く必要すら無い (簡単かつ汎用化できる)
- パイプ
  - 単機能なプログラムを組み合わせて高機能を作り出せる
  - 単機能 : 再利用しやすい , デバッグしやすい

本実験におけるパイプ

- 後に，sox という録音再生ツールと，自分で作るプログラムをパイプでつないで電話を作る
- さしあたり「録音再生」は sox コマンドにまかせる

```
$ 録音コマンド | 自分の電話プログラム | 再生コマンド
```

sox におまかせ

補足 : fopen 系の標準入出力

- もちろん正体は同じものだが，ファイルディスクリプタ (int) とファイル構造体 (FILE *) の表面上の違いから，見た目が異なる
  - 標準入力 : stdin
  - 標準出力 : stdout
  - 標準エラー出力 : stderr
- よって例えば
  - fwrite(data, s, n, stdout) ≈ write(1, data, s*n)
  - fread(data, s, n, stdin) ≈ read(0, data, s*n)