8 ソフトウェア工学
Software Engineering

プログラミング作法

PRACTICE OF PROGRAMMING

プログラミング作法とは

- プログラミングの理論ではない
- プログラミングの細かな実践的ノウハウ
- ソフトウェア工学的な価値観
- 今回学ぶ内容: スタイル, テスト, デバッグ, 移植性

【参考文献】
1) B. W. Kernighan, R. Pike: プログラミング作法, アスキー(2000)
2) D. Boswell, T. Foucher: リーダブルコード, オライリー(2012)

STYLE

スタイル

グローバル変数の名前はわかりやすく

■グローバル変数の宣言

```
int n = 0;
```

```
int nusers = 0;    // ユーザの人数
```

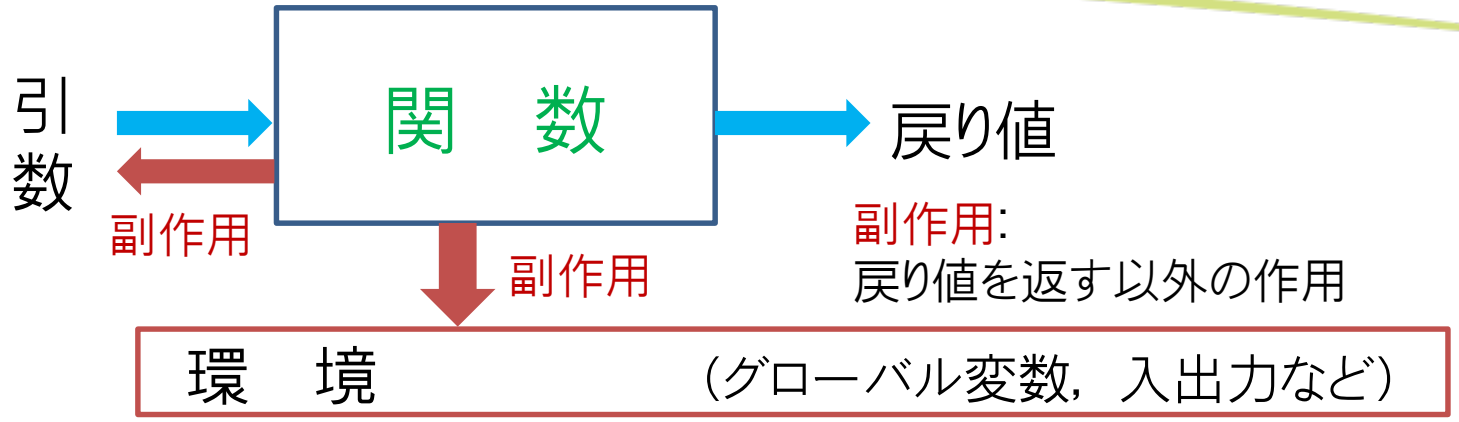
ローカル変数の名前は短めに

■ローカル変数

```
int theElementIndex; // 要素の添え字

for (theElementIndex = 0; theElementIndex < nusers;
                          theElementIndex++)
  elementArray[theElementIndex] = theElementIndex;
```

```
int i;

for (i = 0; i < nusers; i++)
    elem[i] = i;
```

関数名の付け方は統一的に



【統一的な命名規則の例】

- 副作用が主のもの:　作用を表す動詞
  add(canvas, pic),　print(str)
- 戻り値が主のもの:　戻り値の意味を表す名詞またはget+名詞
  length(str),　getTime(date)
- 真偽値を返すもの:　形容詞または is+名詞
  even(n),　isDigit(ch)

複雑な式は分割しよう

```
*x += (*xp=(2*k < (n-m) ? c[k+1] : d[k--]))
```

```
if (2*k < n-m)
    *xp = c[k+1];
else
    *xp = d[k--];
*x += *xp;
```

関数にコメントを

```
/*
 * idct: 2次元8×8逆コサイン変換
 * Chen-Wangアルゴリズム(IEEE ASSP-32, pp.803-816, 1984)
 */
Static void idct(int b[8*8])
{ . . .
```

```
// random: [0..r-1]の範囲の整数を返す
int random(int r)
{
    return (int)(Math.floor(Math.random()*r));
}
```

グローバルデータにコメントを

```
int nusers = 0;    // ユーザの人数
```

```
struct Nvtab {          /* 名前(Name)と値(Value)の対応表 */
    int nval;           /* 値の現在の個数 */
    int max;            /* 割り当て済みの値の個数 */
    Nameval *nameval; /* 名前ー値ペアの配列 */
} nvtab;
```

int 型より列挙型が良いことも

列挙型 (enumeration type)

```
typedef enum{INSERT, UPDATE, DELETE} Status ;

Status status = UPDATE;

void operate(Status status) {
    switch(status) {
    case INSERT:
        printf("登録します。");
        break;
    case UPDATE:
        printf("更新します。");
        break;
    case DELETE:
        printf("削除します。");
    }
}
```

enum型を定義し，適切な型名を付ける

変数の宣言と代入

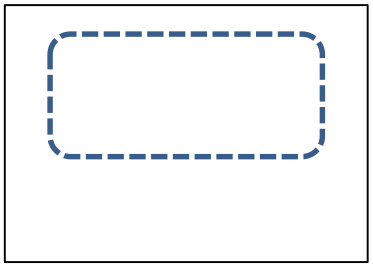
仮引数での宣言

int型のように使える

bool 型より列挙型が良いことも

```
setBoxStyle(false, false, true);
```

```
setBoxStyle(DOTTED, CURVED, TOP);
```



```
typedef enum{SOLID, DOTTED} LineType;
typedef enum{SHARP, CURVED} Corner;
typedef enum{TOP, BOTTOM}   Place;

void setBoxStyle (LineType t, Corner c, Place p){
     . . .
};
```

数値はマクロより定数で定義しよう

```
#define WIDTH 80
#define HEIGHT 24
```

マクロ: ソースコードの文字列を置き換える

C

```
enum {
    WIDTH = 80,
    HEIGHT = 24
};
```

定数: 値を変更できない変数
（コンパイラが管理）

C++

```
const int WIDTH = 80, HEIGHT = 24;
```

Java

```
static final int WIDTH = 80, HEIGHT = 24;
```

変数のスコープはできるだけ狭く

```
int i;

void fun(){
  for(i=0; i<N; i++){
    . . .
  }
}

void main(){
  . . .

}
```

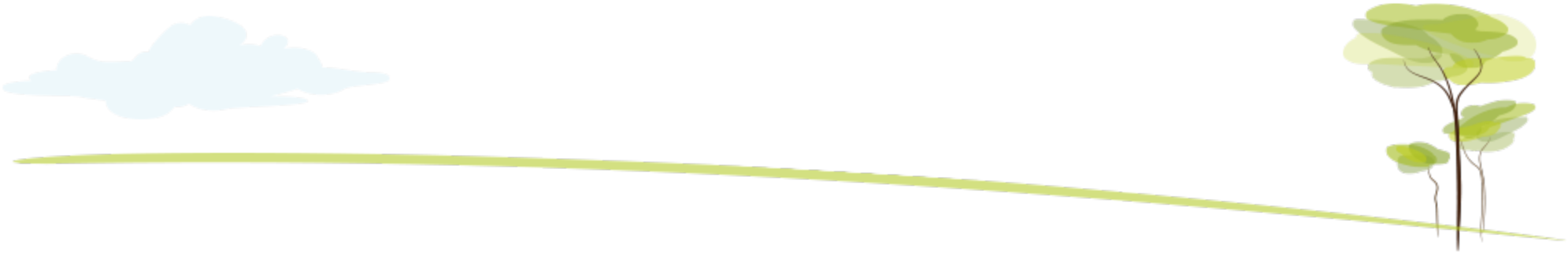
グローバル変数

```
void fun(){
  int i;

  for(i=0; i<N; i++){
    . . .
  }
  . . .
}

void main(){
  . . .
}
```

ローカル変数（関数内で有効）

```
void fun(){
  for(int i=0; i<N; i++){
    . . .
  }
  . . .
}

void main(){
  . . .

}
```

ローカル変数（for文内で有効）



TESTING

テスト

テストの要点

- テストとは: ソフトウェアを動作させて欠陥を発見すること
  （正常と思われるプログラムを破綻させようとする）
- テストでバグの存在は証明できる. バグの不在は証明できない.
- テストは系統的に実行すべき
- 自動化できるものは自動化する

境界をテストしよう

境界条件テスト

ほとんどのバグは境界で発生する

- 入力の境界
  0個(EOF), 1個, 改行のみ

- データ構造(配列など)の境界
  要素がない, データが満杯

- 条件分岐の境界:
  x >= y や x > y の判定で x==yのとき.

事前条件をテストしよう

例: n個の数の平均

```
double avg(double a[], int n) {
    int i;
    double sum;

    sum = 0.0;
    for(i = 0; i < n; i++)
        sum += a[i];
    return sum / n;
}
```

```
assert( n > 0 );
```

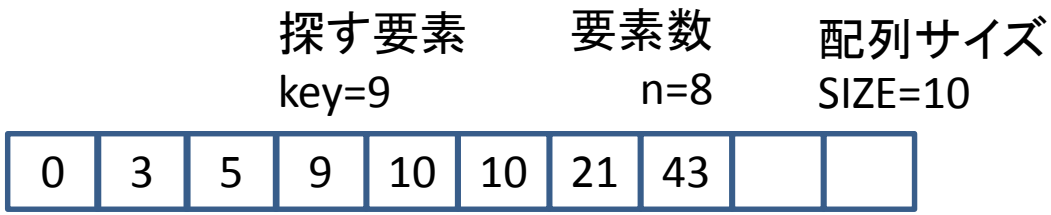
n == 0 でテスト → バグ発見

アサーションを挿入

テストは系統的におこなおう

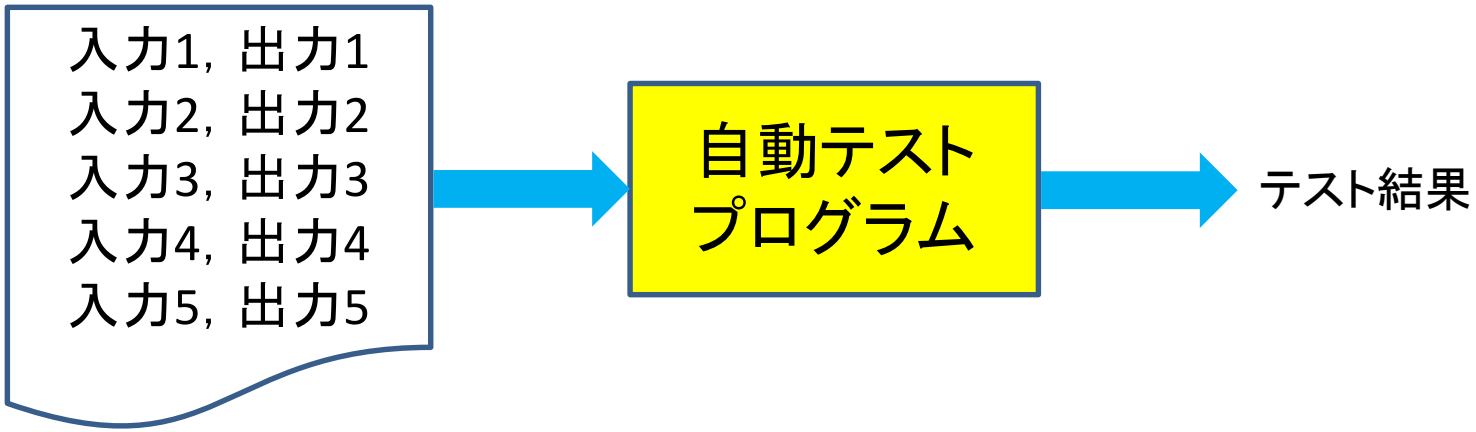
- テストは単純な部品から
- テストは簡単なケースから
- テストはできるだけ網羅的に

【例】二分探索



- n=0 のケース
- n=1(要素 d)で,key<d, key=d, key>d の各ケース
- n=2 で考えられる5種類の全ケース
- n=3 で考えられる7種類の全ケース
- n=2,3,4 で配列に重複した要素dが含まれるとき,
  key<d, key=d, key>d の各ケース
- n=SIZE のケース

テストを自動化しよう



テスト自動化ツールを利用してもよい

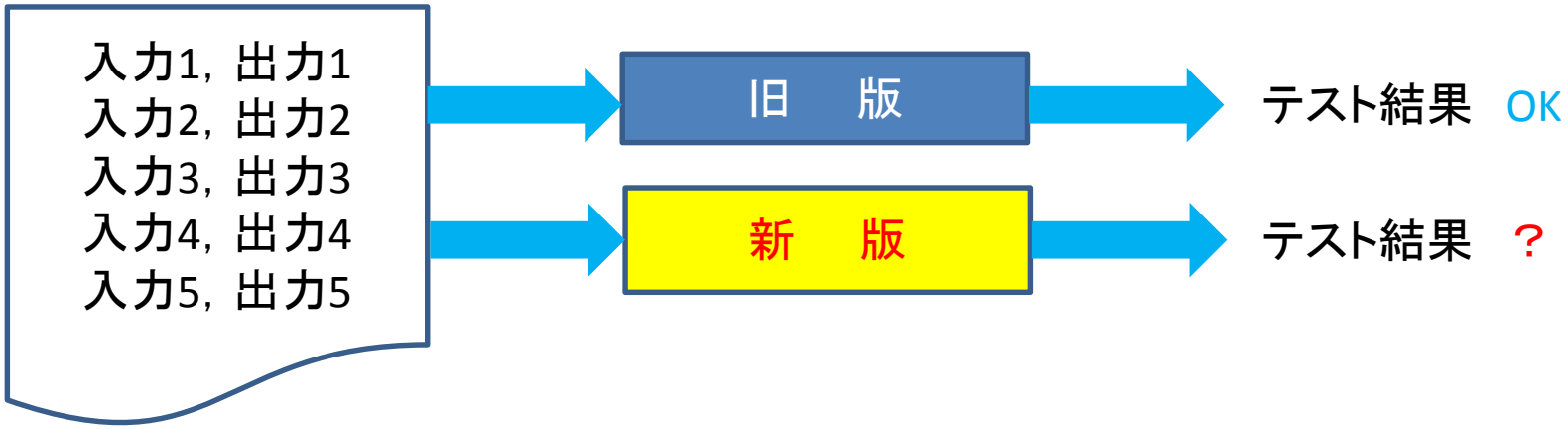
回帰テストを自動化しよう

回帰テスト：
(regression testing)

プログラムに変更を加えた際、それによって
新たな不具合が起きていないかを検証するテスト

旧版の動作が新版でも正しく保たれているか

修正個所とは別の機能が動作しなくなる現象（リグレッション）が生じることがある



ストレステストをおこなおう

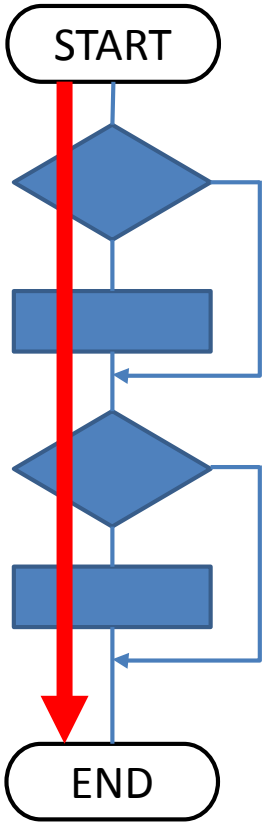
ストレステスト (stress testing)： 要件で定義した限界，または，それを超えた条件で膨大な入力負荷を与えるテスト

【例】 入力バッファや配列のサイズを超える入力
カウンター(int)のオーバーフローを引き起こす入力
短時間に大量のWebアクセス

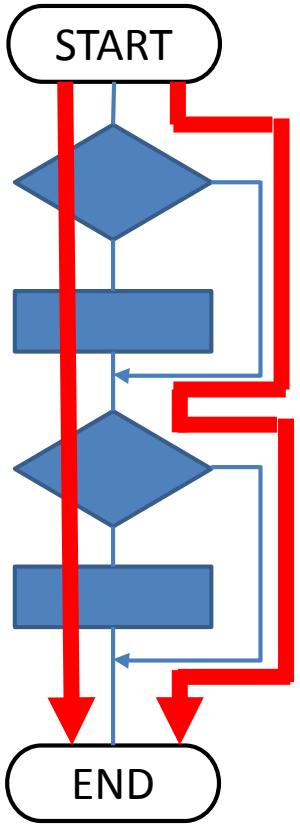
できるだけ網羅的にテストしよう

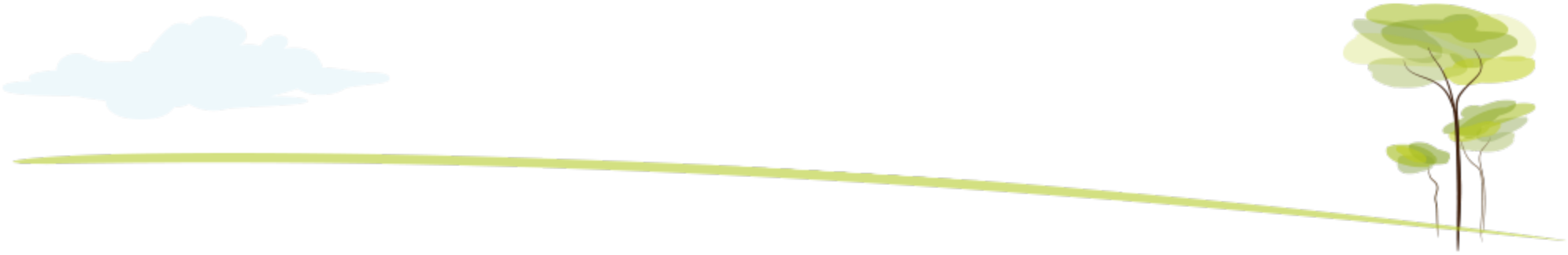
$$\text{網羅率 (coverage)} = \frac{\text{網羅した要素の数}}{\text{網羅したい要素の総数}}$$

命令網羅



分岐網羅

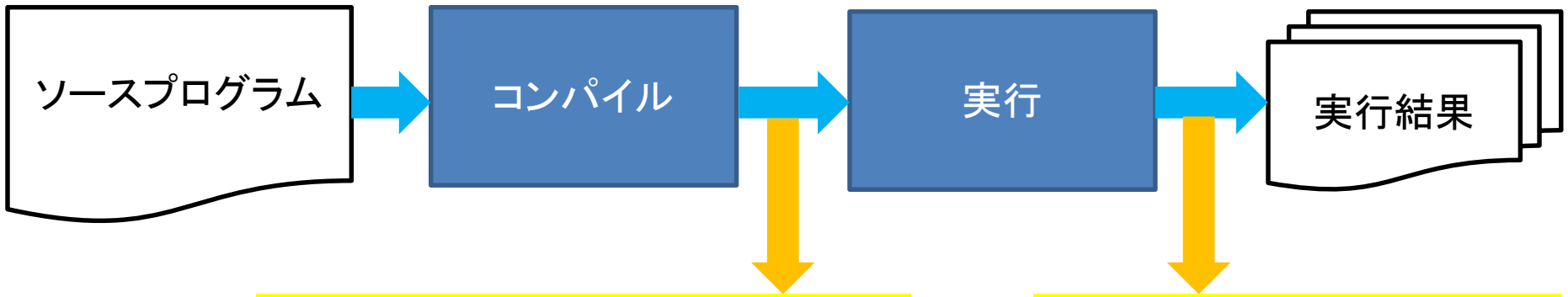




DEBUGGING

デバッグ

コンパイルエラーと実行時エラー



コンパイルエラー
おもに文法的な間違い
　名前のタイプミス
　区切り記号(コンマやカッコ)がない
　実引数と仮引数の不一致

デバッグは比較的容易

実行時エラー
思った通りに動いていない
　変数の初期化ミス
　ポインターの扱いのミス
　メモリのオーバーフロー

デバッグは困難なときも

言語によりエラーの検出力は異なる

自由に書けるが誤りは検出しにくい

同じ内容

C++
プログラム

C++
コンパイラ

実行

コンパイルエラー少ない

実行時エラー多い

Java
プログラム

Java
コンパイラ

実行

制約は多いが誤りは検出しやすい

コンパイルエラー多い

実行時エラー少ない

プリント文で変数の値を確認するのが基本

```
int f(int n){
  if(n==0) return 1;
  else return n*f(n-1);
}
```

```
int f(int n){
  int v;
  printf(”enter f: %d¥n”, n);

  if(n==0) v=1;
  else v=n*f(n-1);

  printf(”return f: %d¥n”, v);
  return v;
}
```

データ数を少なくして配列をプリント

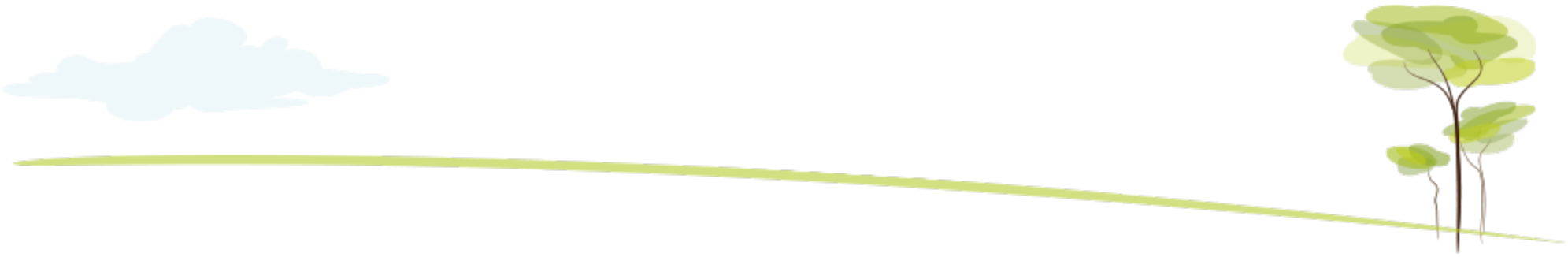
```
# define SIZE 1000
int a[SIZE];

   ...
```

```
# define SIZE 2
int a[SIZE];

   ...

for(i=0; i<SIZE; i++) printf(”%d ”, a[i]);
printf(”¥n”);
```

最後の手段はデバッガ

デバッガ ： デバッグを支援するソフトウェア
debugger

- 実行停止時点での関数呼び出しの列（スタックトレース）を検証
- ローカル変数とグローバル変数の値の表示
- ブレークポイントを設定して，そこまで実行
- 一度にワンステップずつ再実行
- ただし，必ずしも使い易いわけではない



PORTABILITY

移植性

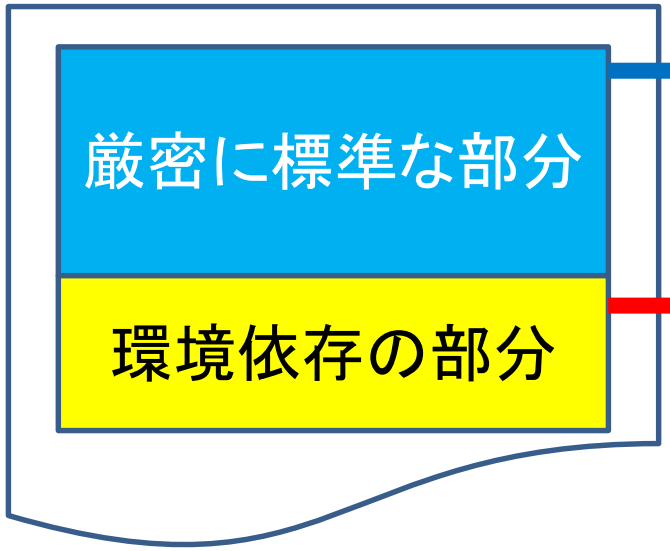
移植性とは

(portability)

1つの環境(コンパイラ, OS, CPUなど)で動くプログラムが
少ない修正で別の環境でも動くこと

現実には, 同じプログラミング言語で書いても動作が環境に依存する

プログラミング言語の環境依存性



できるだけ標準に固執して書けば
移植性が高まる

- コンパイラ依存
- システム(OS)依存
- ハードウェア(CPU)依存
- 文化依存

●言語仕様も改訂される場合がある
●標準が技術的に古く, 現代では不適切な場合もある

コンパイラ依存性

【例】C の char 型は，コンパイラ依存
- 符号つき整数（−128〜127）か 符号なし整数（0〜255）か無規定
- 8ビットかどうかすら規定がない

```
#define EOF -1
char ch;

if((ch = getchar()) == EOF) break;
```

char型が符号なし: ファイル終端で ch==255 となりEOF判定失敗
char型が符号つき: データ 0xFF と EOF（ともに −1）が区別不能

```
int ch;
```
とすべき

Java の char 型は，コンパイラ非依存
- 16ビット符号なし整数

システム依存性

【例】テキストファイルの行末の符号はOS依存

- Windows:　　復帰(CR: '¥r', 0x0D) + 改行(LF: '¥n', 0x0A)
- Unix (Linux): 改行(LF: '¥n', 0x0A)

テキストファイル

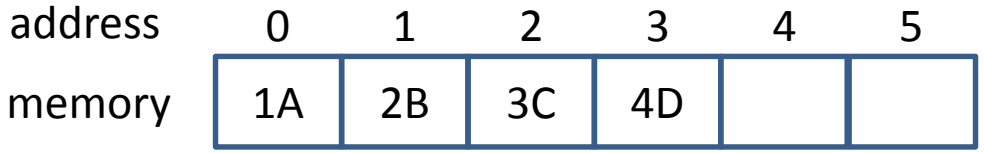
```
AB
12
```

Windows	41	42	0D	0A	31	32	0D	0A
Linux	41	42	0A	31	32	0A		

- CRLFで統一: 入力時には ¥r を除去, 出力時には ¥r を追加
- ファイル形式を変換するプログラムを用意する

ハードウェア依存性

【例】基本データのメモリへの配置: バイト順はCPU依存
- ビッグエンディアンマシン: 下位バイト＝高いアドレス
- リトルエンディアンマシン : 下位バイト＝低いアドレス




```
int n; /* 32ビット */
```

Big endian: n = 0x1A2B3C4D

Little endian: n = 0x4D3C2B1A

➡ int をバイト列で書き(または送信),
別なコンピュータで int として読む(または受信)のは安全でない

文化依存性

【例】文字コード
- ASCII コード
- JIS コード
- Latin-1 コード
- Unicode

【例】日付形式
- 2010年11月12日
- 2010/11/12
- 11/12/10

環境依存部分はインタフェースの裏に隠蔽

APIを使うと
移植性が高い

API: Application Programming Interface

showWindow(w)

showDate(d)

(interface)
インタフェース

分　離

環境依存の実装コード

実装は隠蔽

環境ごとに
依存部分を
実装

Javaは仮想マシンで移植性を高めている

javaファイル

Javaコード

javac コマンド

Javaコンパイラ

classファイル

Java
バイトコード

java コマンド

Java仮想マシン

JVM (Java Virtual Machine)

分　　離

環境依存の JVM 実装

実装は隠蔽

環境ごとに
依存部分を
実装

Windows
Linux
MacOS

演習問題 8

(1) 今回の内容のうち,「スタイル」の作法について,あなたのこれまでのプログラミングスタイルに照らし合わせて,感想を簡単に述べなさい.

(2) あなたが過去に作成したプログラムを1つ想定し,その概略を説明し,そのプログラムのテスト方法の概略を設計しなさい.具体的にどのような入力データを使って,どのようにテストしたらよいかを簡単に述べること.

(3) 前問(2)のプログラムを実行したときに,時間がいつまでたってもプログラムからの出力が得られなかった場合,どのようにデバッグを行ったらよいかを考え,簡単に説明しなさい.