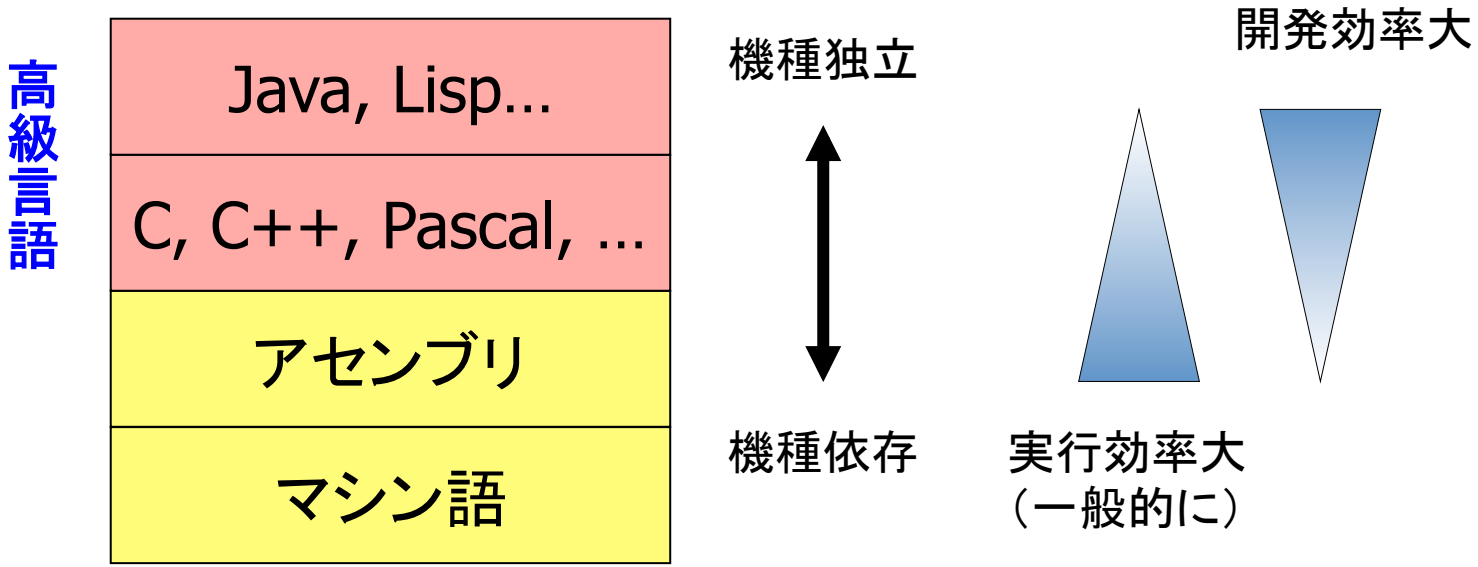
導入

プログラミング言語って何?

- コンピュータに対する処理の指示

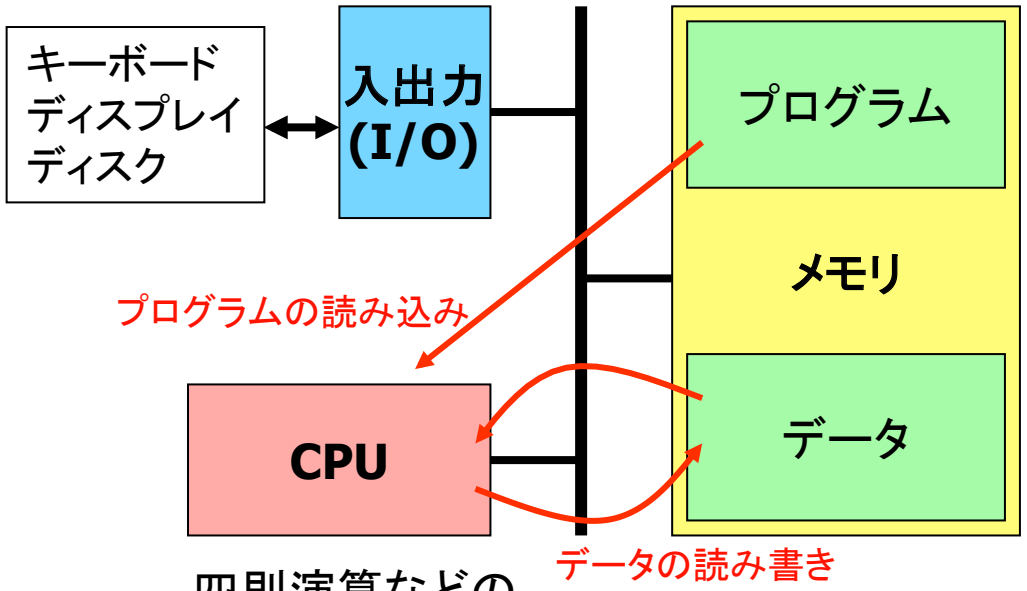


C言語の特徴

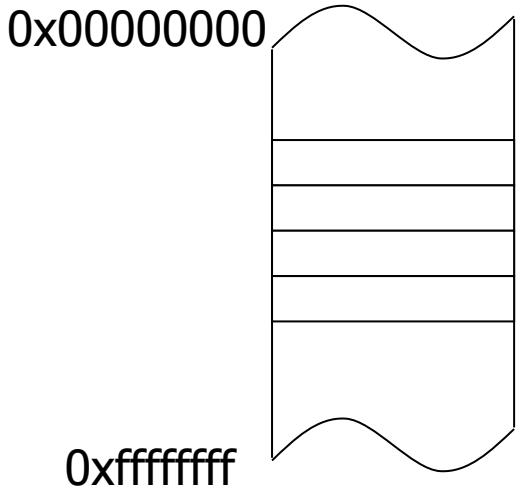
- 分岐、繰り返し等の制御構造
  - 構造化プログラミング
- ポインタ、メモリのアドレス操作が可能
  - ハードの制御を書くことも可能
- Java等の他の言語を理解する上でも共通の基本部分は大きい
  - 最近はハードウェア(回路)の設計も出来る

コンピュータの基本的な仕組み

ノイマン型コンピュータ



四則演算などの
計算、実行制御



メモリはデータを入れる箱の集まり。
各々の箱には番地が付いていて、
好きな場所に読み書きできる。

実際の処理手順

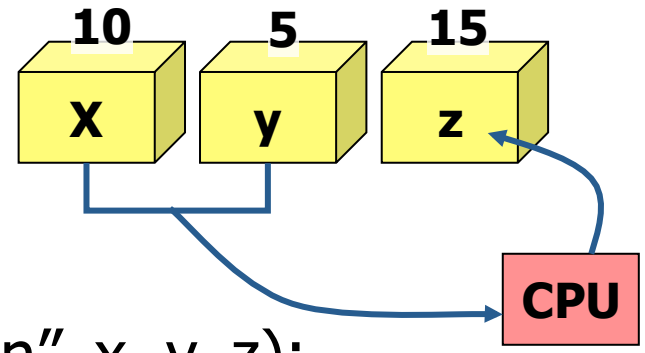
- コンパイル(cc または gcc)
  - Cプログラムを実行コード（マシン語）へ変換
- ソースプログラム(.c)から実行コード

- -oオプションでa.outの名前を変更可能
  - cc –o foo foo.c

変数

- プログラム中で数値などを覚えておく箱
  - 宣言してから使う
- int x;と書くとxという名前の整数を入れる箱ができる

```
int main() {
        int x, y, z;
        x = 10; y = 5;
        z = x + y;
        printf("%d + %d = %d\n", x, y, z);
        return(0);
}
```



関数の作り方

- 処理のまとまりを関数として定義できる
  - () の中の変数は仮引数と呼ばれる
  - 呼び出される関数を先（上）に書く

仮引数：呼び出された際に引数が代入される

```
void f(int x, int y) {
        printf("%d + %d = %d\n", x, y, x+y);
        printf("%d * %d = %d\n", x, y, x*y);
}

int main() {
        f(2, 4);
        return(0);
}
```

結果を返さない関数は最初をvoidとする

結果を返す関数

- 計算した結果を返す場合returnを使う

返す結果の値の種類に合わせる

```
int sanjo(int x) { /* 3乗を計算する関数 */
        return(x*x*x);
}

int main() { /* 最初に呼び出される関数 */
        int x;  /* 変数宣言 */
        x = sanjo (7); /* 3乗の値の計算と変数への代入 */
        printf("7 の3乗は %d です。\n", x);
        return(0);
}
```

補足:コメント文

- プログラム中で /* と */に囲まれた部分
- コンパイル時にはないものとして処理

- 処理の目的などを書いておく
  - 後から見て、わかるようにする

変数と演算子

変数と演算子

- C言語での決まり
  - コンパイラが解析・解釈できる規則
- 字句
  - 識別子、予約語（キーワード）、定数、文字列
  - 区切りと空白
- 変数と型
- 演算子

字句：識別子

- 変数や関数の名前に使う
- 英字(アルファベットと_)で始まり、任意の数の英数字(英字と数字)が続く
  - つまり、数字で始まってはいけない
  - 大文字と小文字は区別される

    例) temp Temp R2D2 my_name

- 予約語は識別子として使ってはならない
  - return や int など
  - printf は関数名であり予約語ではない

字句:定数

- 整数
  - 普通の数字は10進
  - 0で始まると8進、0xで始まると16進
    - 011 (= 9), 0x11 (=17)
- 小数
  - 実数は浮動小数点数(floating-point)であらわす
    - 誤差がある
  - 小数点表記と指数表記
    - 314.142, 3.14142e2, 3.14142E2
  - 内部的には仮数部と指数部で表現される

字句：文字定数・文字列定数

- 文字
  - ’ (シングルクォート)で囲む
  - 特殊文字(改行…’\n’, タブ…’\t’)
  - \ は ‘\\’, シングルクォートは ‘\”
- 文字列
  - “ (ダブルクォート)で囲む
  - 特殊文字は文字の表記に準ずる
  - ダブルクォートは \”

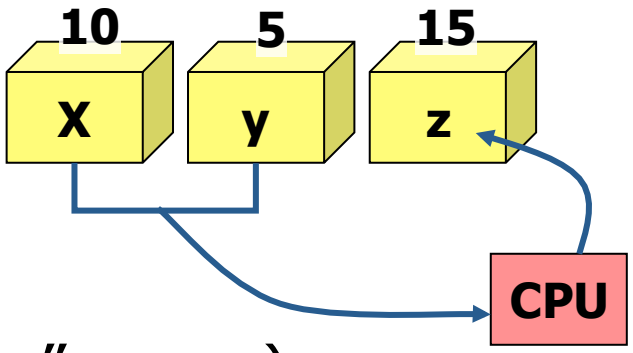
字句：区切りと空白

- ; (セミコロン)
  - 文の区切り
  - 改行・空白が自由な分に区切りは大事
- , (カンマ)
  - 引数の区切りなど
- { }
  - 複数の文をひとまとめにする
- 空白
  - スペース、タブ、改行は数で挙動は変わらない

変数と型

- プログラム中で数値などを覚えておく箱
  - 宣言してから使う
- int x; と書くと x という名前の整数を入れる箱ができる

```
int main() {
        int x, y, z;
        x = 10; y = 5;
        z = x + y;
        printf("%d + %d = %d\n", x, y, z);
        return(0);
}
```



変数と型(1)

- 箱には入れる物によって種類がある
  - int　… 整数
    - サイズは処理系依存(最も効率の良い長さ)
      - 最近のマシンでは32bit(=4byte)が多い
    - 前にshort, longをつけることが出来る
      - short int (short) ≦ int ≦ long int (long)
  - char　… 文字
    - 1byte (-128～127 or 0～255)
    - 正の数だけか負の数も含むかは実装依存

変数と型(2)

- float
  - 単精度浮動小数点：多くの場合32bit
- double
  - 倍精度浮動小数点：多くの場合64bitでfloatより誤差が小さい
- 変数のサイズ（バイト数）はsizeof関数に型名や変数名を渡すとわかる

```
int x; x = sizeof (double);
```

大量のデータを扱うなど、特殊な理由がある場合以外は、整数にint、小数にdoubleを使う

自動（auto）変数

- 関数の中で普通に宣言した変数
- その関数の中でのみ有効
  - 関数が呼ばれたときに割り当てられる
  - 関数から戻るときにはなくなる
- 通常は，記述は省略されている

auto int n; → int n;

グローバル変数(大域変数)

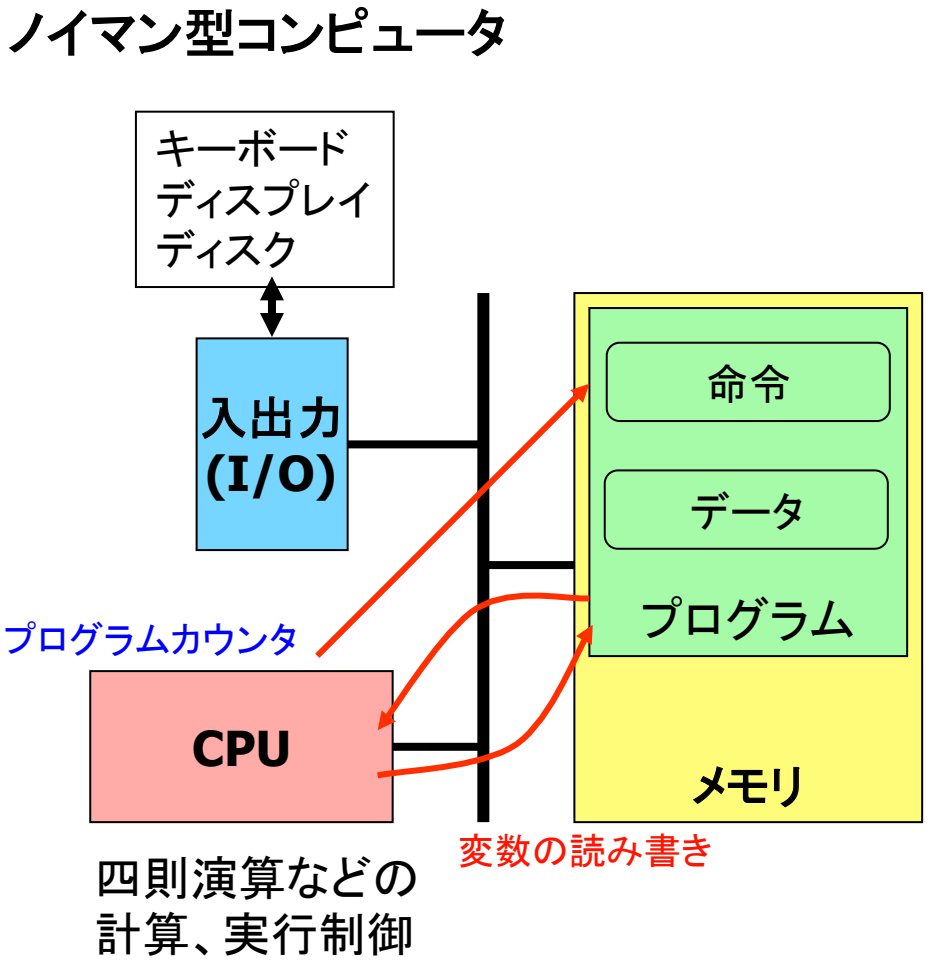
- 関数の外で変数宣言
- そのファイル中の全ての関数から使える
- 便利だが、必要な場合以外は使わない
  - プログラムが読み難くなる
  - 変数の役割によって使い分ける
- ファイル間で大域変数を相互参照できる
  - 参照するファイルの中で extern 宣言する
  - 例： extern int system_id; /* system_id は、他のファイルで大域変数として宣言されている */

変数の使用例

```
int system_id; /* 大域変数 */

int checkId (int n, int m) {
    int sys_id = 10 * n + m; /* 局所変数 */
    if (sys_id != system_id)
        return 0;
    else
        return 1;
}

void changeId (int n) {
    system_id = n;
}
```

コンピュータの実際

- コンピュータはどうやって動いている？
  - 命令もデータも同じメモリ上にある
  - 命令を一つずつ進め、命令に応じてデータの部分にアクセス

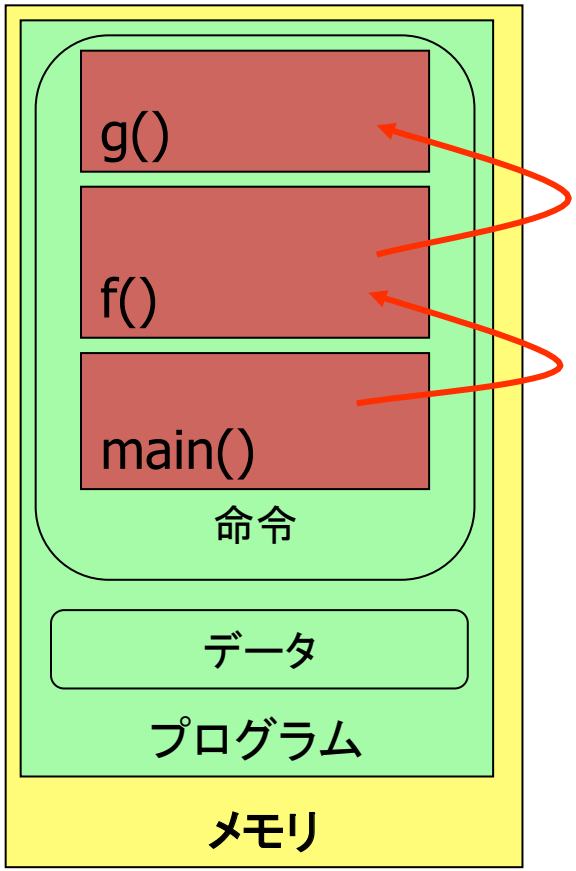


関数の実際

- 関数を定義して呼び出す

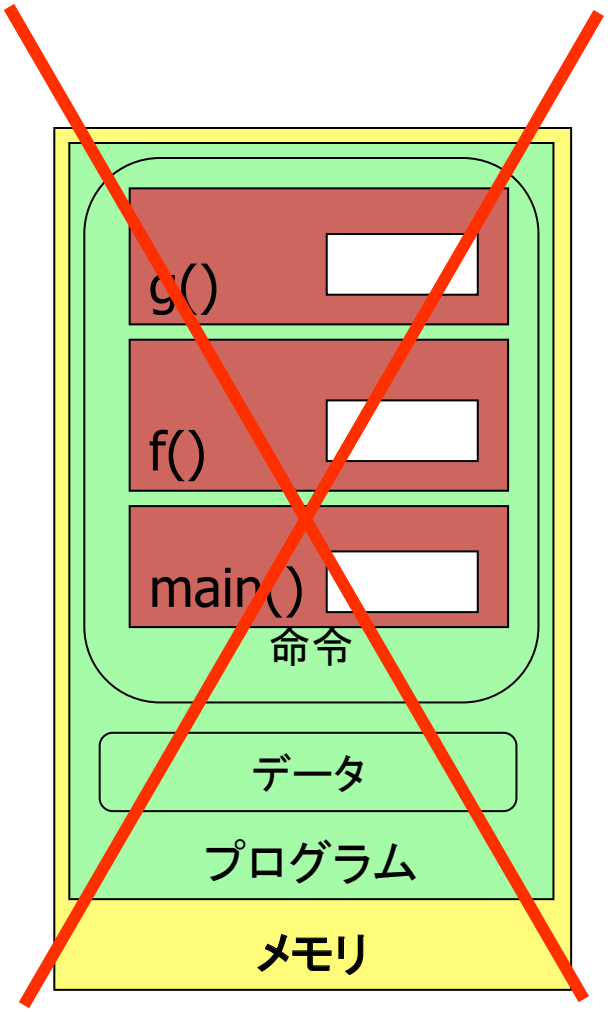
```
main(void) {
        ...;
        f(x);
}

int f(int x) {
        ...;
        g(x);
}
```

```
int g(int x) {
        ...;
}
```



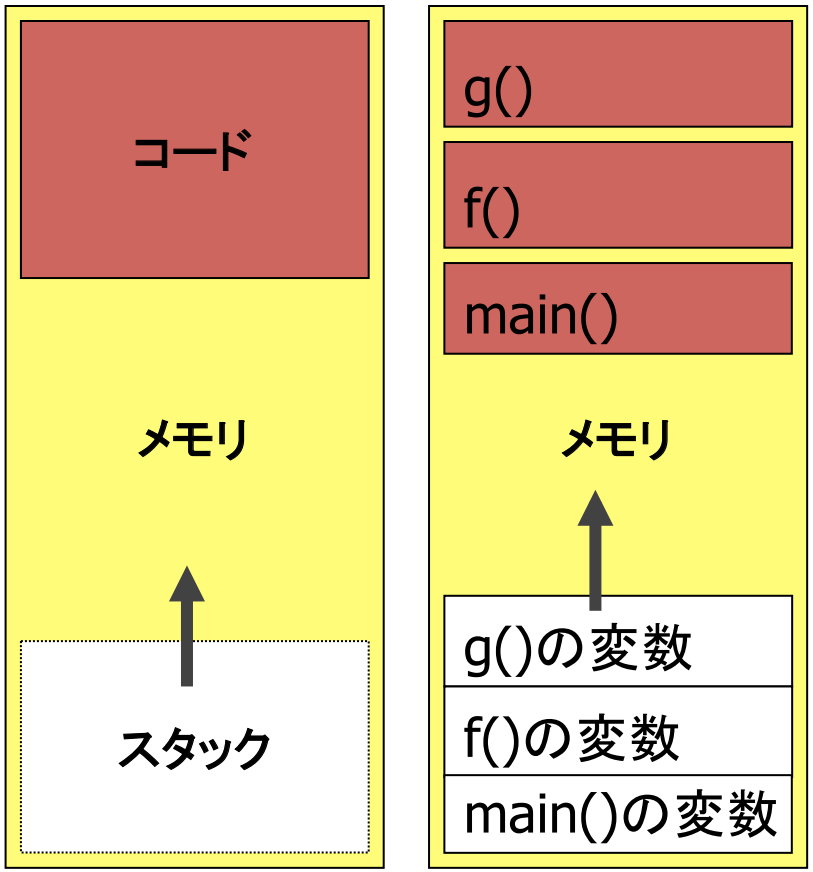
変数割り当ての実際

- auto変数は何処にある？
  - 関数の中でだけ有効
  - 関数の命令毎にある？



関数呼び出しの実際(1/2)

- 関数中でのみ有効なデータの領域
  - 関数が**呼び出されるごとに確保**される
    - 実行時の環境
  - スタック領域
  - コード領域



関数呼び出しの実際(2/2)

- auto変数 x は呼ばれ方によって記憶場所が違う

```
void f() {
        int x;
        printf("addr of x in f() is %p\n", &x);
}
void g() {
        int x;
        printf("addr of x in g() is %p\n", &x);
}
int main(void) {
        f();
        g();
}
```

実行結果：

```
addr of x in f() is 0xbfbff5c
addr of x in g() is 0xbfbffac
```

演算子

- 単項演算子
  - オペランド（演算対象）が一つ
  - 前置と後置がある
- 2項演算子
  - オペランドが二つ
- 算術演算子
  - +, ー, *, /, %(割り算結果の余り)

代入演算子

- =(左辺に代入)
- +=, ー=, *=, /=, %=, &=, |=, ^=(算術・ビット演算をした結果を左辺に代入)
  - x += 1; ⇔ x = x + 1;
  - x &= 0x11; ⇔ x = x & 0x11:
- 代入演算子の結果は代入した値
  - y=x=z + 10 とすると、z+10の結果がxに代入され、さらにyに代入される

インクリメント・デクリメント演算子

- ++(1増やす), −−(1減らす)
- x = x +1; x = x − 1; の簡易表現
- 前置すると式の評価前に加減、後置すると評価後に加減

```
x=1;
a = ++x; /* aには2が代入される */
b = x++; /* bには2が代入される */
```

関係演算子と論理演算子

- 真(1)か偽(0)
- 条件分岐（if 文）で使用される
- <, >, <=, >=, ==, !=
- &&(論理積), || (論理和), ! (論理反転)
  - 適用の優先順位: ! → && → ||
  - ( ) を用いて優先順位を任意に変更できる

if (a>0 || b && c==1) は if (a>0 || (b && c==1)) と同じ意味

- = と == を区別せよ！
  - =は代入なので、代入値0以外では必ずtrue
- 論理演算子の結果は数字

型演算子

- 数値の型を変換する
- 代入される数値の前に型名を()で囲む

```
int a=3;
double f;
f = (double) a; /* 3.0が代入される */
a = (int) 4.123; /* 4が代入される */
```

型変換が必要なケース

- 整数　→　浮動小数点
  - 計算可能な数値範囲を拡大させる
  - 高い計算精度を要求する
  - 数学ライブラリ関数を利用する
- 浮動小数点　→　整数
  - 離散的な演算（回数，順序等）を扱う
  - 計算結果を離散化する（ex. インデックスとして用いる）
  - 計算を高速化する

繰り返しと条件分岐

繰り返し

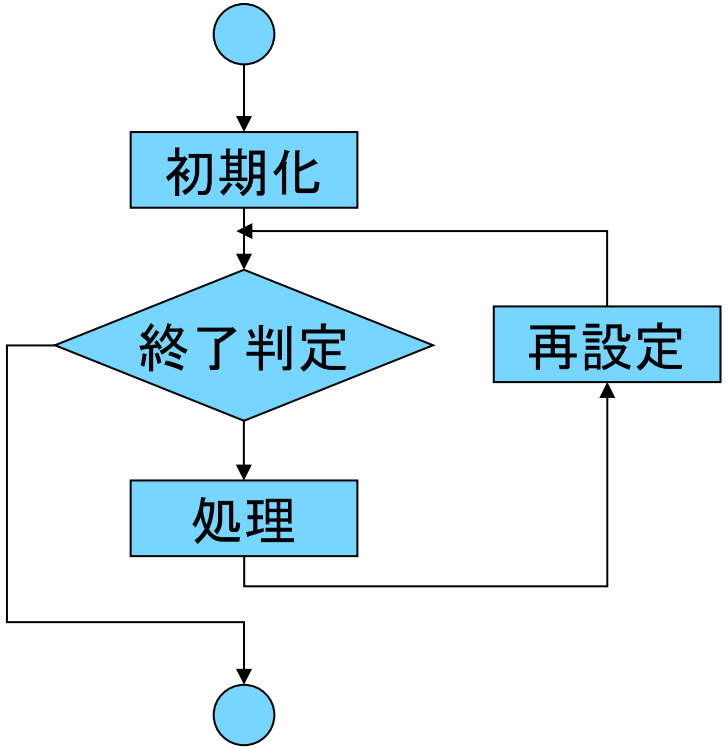
- 処理を繰り返し実行する
  - for文
  - while文
  - do - while文
- ひたすら処理を羅列するより、「ある処理を〇回繰り返す」と書いたほうが読みやすい。
- 処理中に条件判定に関する変数を正しく変化させないと無限に繰り返す

for文

- 繰り返しに関することが一文で書ける

```
for (初期化; 終了判定; 再設定) {
        処理;
}
```

```
int  i, sum=0;
for ( i = 1; i <=10; i++) {
        sum = sum + i;
}
```



繰り返し回数の指定法

- 繰り返し文をどう書くか？

  ５回繰り返したい…

```
for(i=0; i < 5; i++) {
        …;
}
```

```
for(i=0; i <= 4; i++) {
        …;
}
```

```
for(i=1; i < 6; i++) {
        …;
}
```

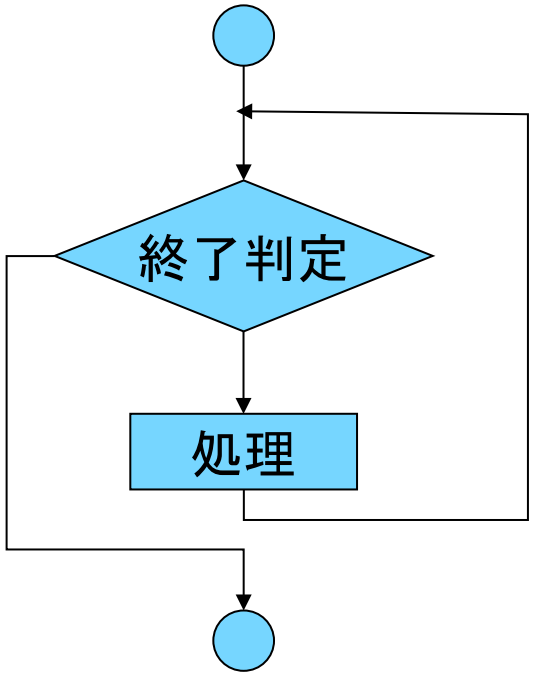
```
for(i=1; i <= 5; i++) {
        …;
}
```

while文

- 条件式が成り立つ間繰り返す
  - まず判定、それから処理

```
while (終了判定) {
        処理;
}
```

```
int  i, sum;
i=1; sum=0;
while (i <= 10) {
        sum = sum + i;
        i = i + 1;
}
```

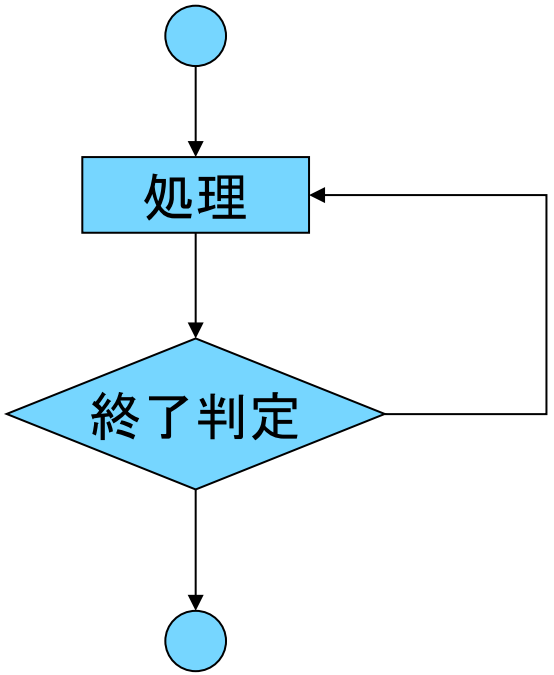


do-while文

- 条件式が成り立つ間繰り返す
  - まず処理、それから判定

```
do {
        処理;
} while (終了判定);
```

```
int  i, sum;
i=1; sum=1;
do {
        sum += i++;
} while (i <= 10);
```



break文とcontinue文

- 繰り返しの処理を途中で抜けたり、
  次の繰り返しに跳んだり、の制御

```
while (終了判定) {
        処理1;
        if (…)
                break;
        処理2;
}
処理3;
```

breakは処理3に跳ぶ
（繰り返しは終了）

```
while (終了判定) {
        処理1;
        if (…)
                continue;
        処理2;
}
処理3;
```

continueは終了判定に跳ぶ
（繰り返しは続く）

条件分岐

- 関係演算子や論理演算子の結果によって命令を切り替える
  - if - else 文
  - switch 文
- 制御の流れを決定する

if 文(1/3)

- 時と場合によって処理を変えたい
  - if ...
  - if ... else ...
  - if ... else if ... else ...

ある場合はAの処理

```
if (条件) {
        処理A;
}
```

if 文(2/3)

- if ... else ...

ある場合はAの処理
それ以外はBの処理

```
if (条件) {
        処理A;
} else {
        処理B;
}
```

if 文(3/3)

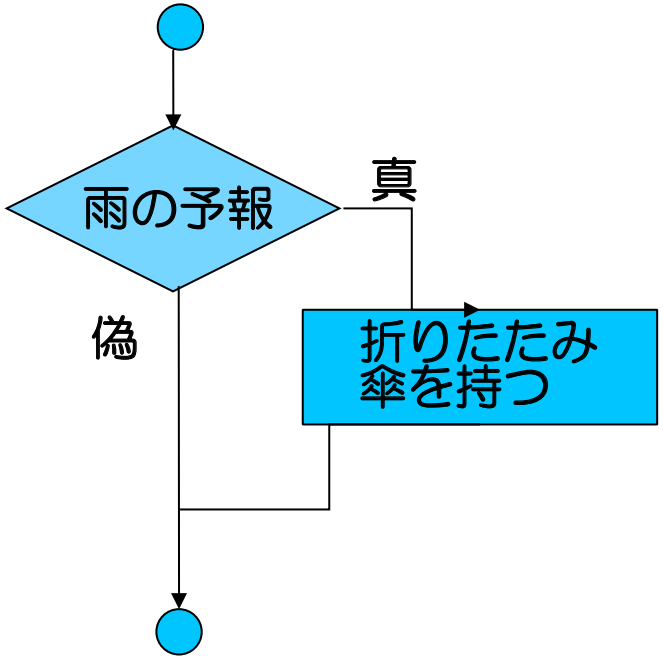
- if ... else if ... else ...

ある場合はAの処理
それ以外のある場合はBの処理
それら以外はCの処理

```
if (条件1) {
        処理A;
} else if (条件2) {
        処理B;
} else {
        処理C;
}
```

if 文の例(1/4)

- 雨の予報なら折りたたみ傘を持っていく.

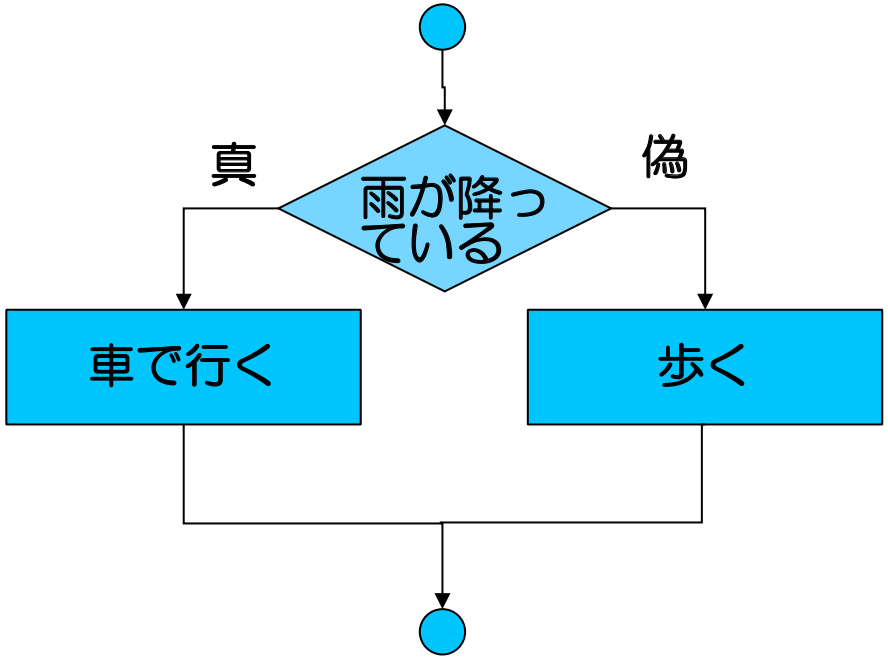



```
if (雨の予報) {
        折りたたみ傘を持つ;
}
```

こういうのを擬似コード
(pseudo code)という。

if 文の例(2/4)

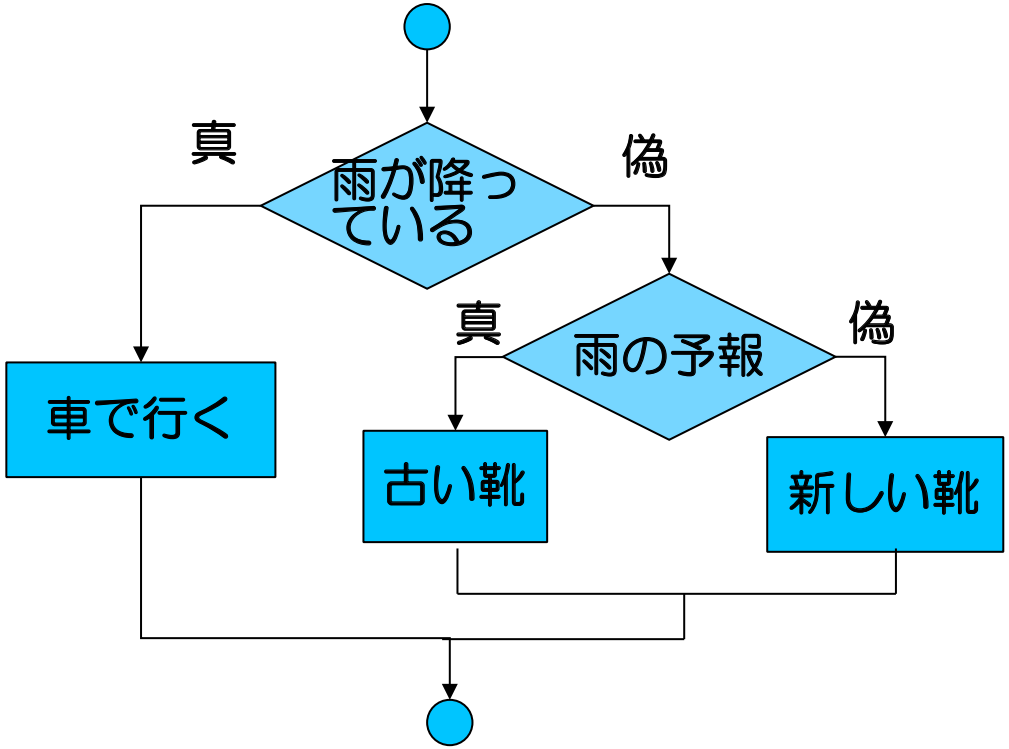
- 雨が降っているなら車で行き，そうでなければ歩く.




```
if (雨が降っている) {
        車で行く;
} else {
        歩く;
}
```

if 文の例(3/4)

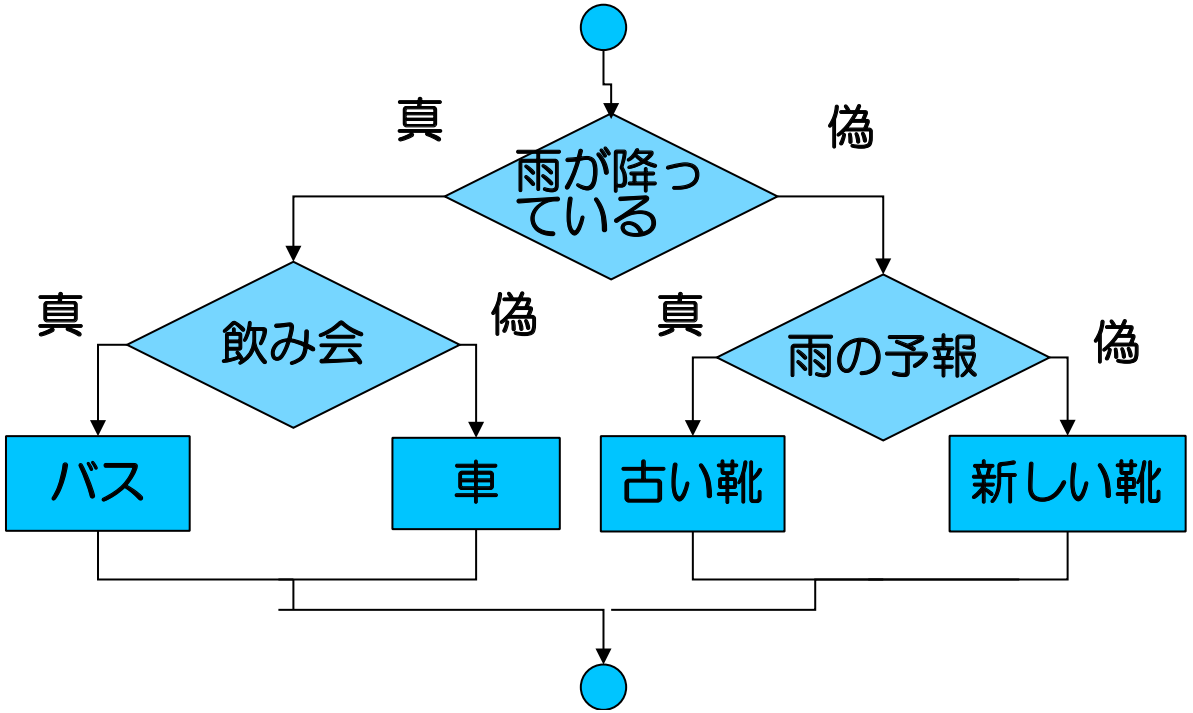
- 雨が降っているなら車でいき，そうでないときに雨の予報なら古い靴を履く．そうでなければ新しい靴を履く．




```
if (雨が降っている) {
        車で行く;
} else if (雨の予報) {
        古い靴;
} else {
        新しい靴;
}
```

if 文の例(4/4)

- 出かけるときに雨が降っている
  - 飲み会があればバスで行く．そうでなければ車で行く．
- 雨が降っていない
  - 雨の予報なら古い靴を履く．そうでなければ新しい靴を履く．

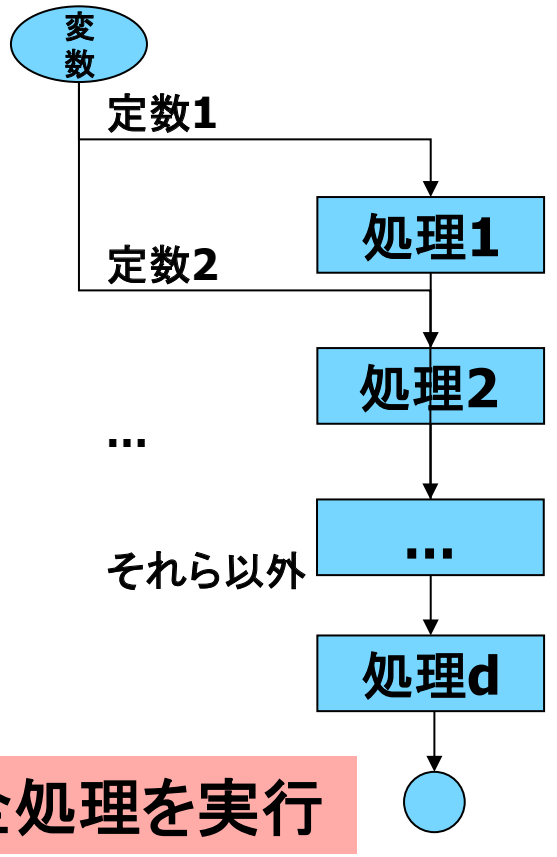



```
if (雨が降っている) {
        if (飲み会) {
                バスで行く;
        } else {
                車で行く;
        }
} else {
        if (雨の予報) {
                古い靴;
        } else {
                新しい靴;
        }
}
```

switch 文(1/3)

- 変数の値によって振り分ける

```
switch(変数) {
        case 定数1:
                処理1;
        case 定数2:
                処理2;
           …
        default;
                処理d;
}
```

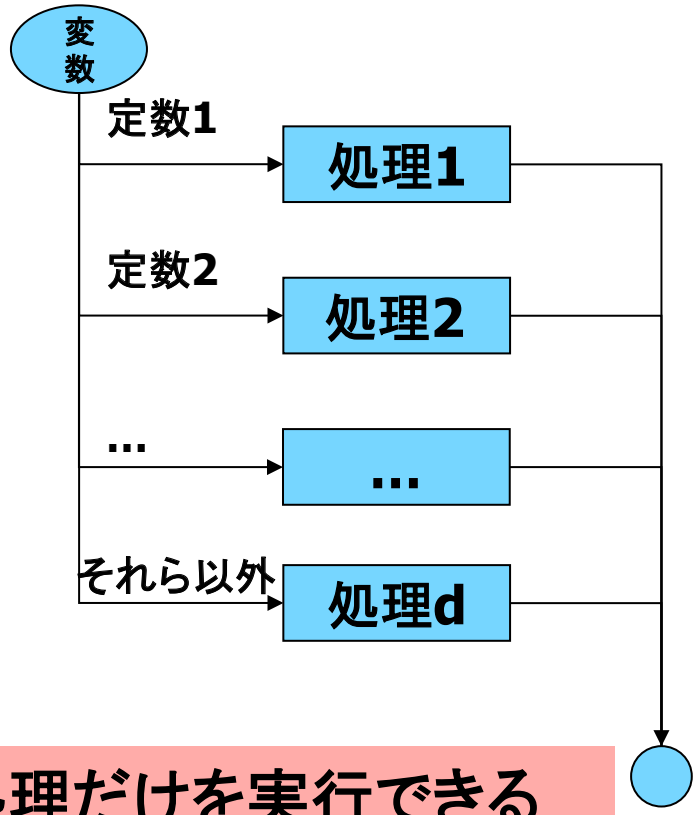


該当する条件以下に記述された全処理を実行

switch 文(2/3)

- breakを使うとそこでswitch文から抜ける

```
switch(変数) {
        case 定数1:
                処理1;
                break;
        case 定数2:
                処理2;
                break;
           …
        default;
                処理d;
}
```

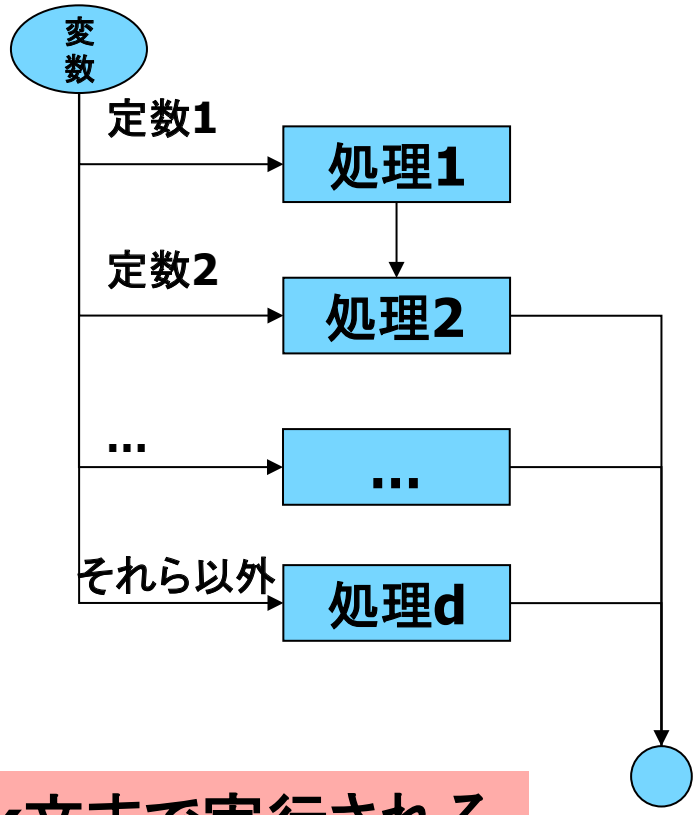


break文を付けると個々の処理だけを実行できる

switch 文(3/3)

- break文は任意の箇所に記述できる

```
switch(変数) {
        case 定数1:
                処理1;
        case 定数2:
                処理2;
                break;
            …
        default;
                処理d;
}
```



該当する条件以下のbreak文まで実行される