2016.12.15改訂

プログラム理論と言語　第4回
—手続き型言語(後編)—

有村博紀
吉岡真治

本スライドは，2015北海道大学　吉岡真治「プログラム理論と言語」に基づいて，現著者の承諾のもとに，改訂者(有村)が加筆修正しています．本スライドに誤り等があった場合には，内容の文責は改訂者にありますのでご注意ください．
2016北海道大学，有村博紀

- 今回は集計なし

- イントロダクション
  - プログラミング言語の歴史
- プログラミング言語と背景理論
  - 手続き型と関数型
  - オブジェクト指向型と論理型
    - 言語の比較を交えつつ

主な参考書(教科書)

- プログラミング言語の概念と構造，新装版，
ラビ・セシィ，ピアソン5600円
- 授業のネタ本です．（わからなくなったら図書館等でみてみよう）

- 学ぶこと
  - 手続き型言語とは（なぜ生まれたか？）
  - その特徴．長所と短所

- ポイント
  - 構造化プログラミング
  - 手続き･関数の定義
  - 変数の有効範囲（スコープ）
  - [発展] 手続き･関数の実行の仕方
  - 値呼びと参照呼び
  - [発展] ポインタの利用とメモリ確保の方法

手続き型言語（命令型言語）

ここは大事！

- 定義
  - 計算機の行う手続きのうち、意味のある塊を取り出して手続きとして定義する。
  - 手続きを呼び出すことによって変数（メモリ）の内容を変えながら計算を行う言語
- もう少し補足すると
  - プログラムを，手順（手続き）の並びとして実現するプログラム言語．プログラムの動作により，変数の内容や出力を変化させることで仕事をする．（メモリや外部への「副作用」をもつ）
    - ⇔関数型言語（関数値しか持たない）
  - 伝統的な言語：例）C言語，Fortran，Algol（Pascal）
  - 命令型言語（imperative language）とも呼ぶ

手続き型言語

ここは大事！

- 次の3つの主要な概念でできている
- 1. 代入
- 2. 書き換え可能なデータ構造
- 3. 制御文

```
while (x != A[i]) {
    i := i – 1;
}
```

手続き型言語

- 代入と制御文を用いて計算を行う

- 代入文
- 複文（文の合成）
- 条件文
  - IF文, IF-THEN文, IF-THEN-ELSE文
  - CASE文
- くり返し文
  - FOR文
  - WHILE文
  - DO-UNTIL文
- 手続き宣言と手続き呼び出し

- その他
  - GOTO文（使わない方が良い）.
  - CATCH-THROUGH文（使うと良い）

手続き型言語：設計思想

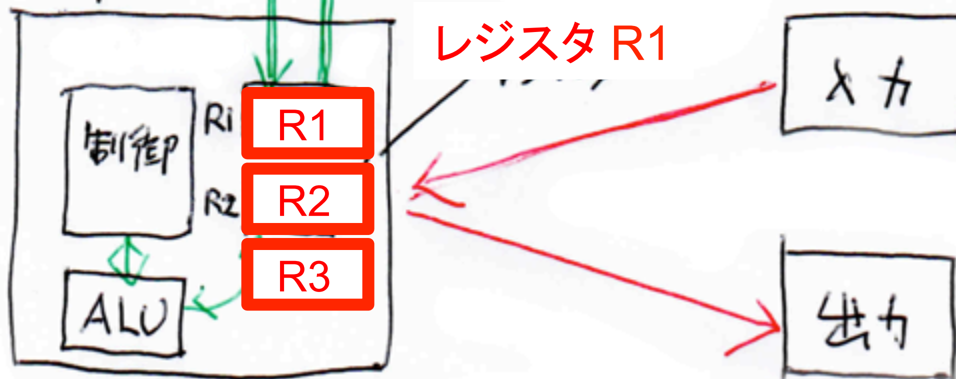
ここは大事！

- マシンモデルに近い
  - 言語は，元となる計算機を，直接的かつ効率的に使用できるように設計されている
- 構造化プログラミング
  - 計算機を機械語で直接プログラミングするのは（人間の）作業効率が悪い
  - プログラム言語は，それが何をするかがプログラムの字面（テキスト）から理解しやすいようになっているべき．

計算機（コンピュータ）のハードウェア（再掲）

メモリ （記憶装置）

変数X (=130番地)

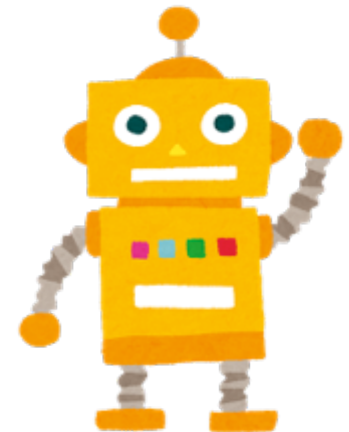


命令ADD R3 R1 R2

機械語：

算術・論理命令
メモリ命令
ジャンプ命令
条件ジャンプ命令

機械の都合に合わせて
プログラミングするのは
人間にはたいへんだ！



これはいわゆるコンピュータの「命令セットレベル」の見方

代入：　左辺値と右辺値

ここは大事！

- プログラムの動きを理解するには，変数の左辺値と右辺値という概念が大事
  - 例）関数呼び出しなど
- 左辺値(L-value)：値の場所(番地)を表す
- 右辺値(R-value)：値の中身そのものを表す

```
int x, y;
x := 2 + 3;
y := x;
```

ここは大事！

変数の種類

- 変数の有効範囲 (スコープscope)
  - プログラム中で各変数が有効な範囲(みえる部分)
  - 例(C言語):各関数で定義される変数(ローカル変数)と関数の外側で定義される変数(グローバル変数)
  - 入れ子の内側が有効
    (C言語:ローカル変数が優先)

- 例(右のプログラム)
  - 副手続きprintx中の局所変数xのスコープは, スコープCの範囲
  - グローバル変数xのスコープは, スコープAの範囲からスコープCの範囲を除いた残りの範囲
    (背景が水色の範囲)

```
#include<stdio.h>
int x;                      スコープA

int main() {                スコープB
  x = 100;
  printx();
  printf("X=%d\n",x);
}

int printx() {              スコープC
  int x;
  x =50;
  printf("X=%d\n",x);
}
```

```
X=50
X=100
```

注1)ここではコンパイラの標準的教科書[ALSU'09]に従って, "scope"を「有効範囲」と訳した. [ALSU'09] A.V.Aho, M.S.Lam, R.Sethi, J.D. Ullman, "Compilers," 2nd, Addison-Wesley, 2009.(いわゆる「ドラゴンブック」)

注2): C言語とほとんどの言語は構文で決まる「静的な」有効範囲を採用している. LISPなどいくつかの言語は実行時に決まる「動的な」有効範囲を採用している.

モジュール性の向上のために

- 記号の衝突
  - ライブラリなどの関数定義中で用いる記号とユーザが利用する記号の関係
    - 同じ記号の変数はすべて同じものとすると、ライブラリを使うユーザは、使われている変数の名前を全て知っている必要があり、モジュール性が低い
- 変数のスコープ
  - ローカル変数とグローバル変数

手続き・関数の定義:C言語

ここは大事！

- 手続き・関数宣言（入出力）
  - 引数:入力の数と各々の変数の型（type）
  - 返り値:返り値の変数型
- 仮引数:関数宣言に書く引数.　例）x, y
- 実引数:関数呼び出しで渡す引数.　例）2, 3

```
//関数宣言
int sum(int x, int y){
  return x+y;
}
```

```
//関数呼び出し
int z;
z = sum(2, 3)
```

手続き･関数呼び出しの実行

- 手続きや関数*は次のように計算機で実行される
- （板書します．）
- （次の4枚で説明します）

注*） 本スライドでは，最近の慣例にしたがい，手続き（procedure/subroutine）と関数（function）をとくに区別しません．そこで，以降では両者を含めて手続きと呼ぶことがあります．また，関数型プログラミングのように，値を返し，副作用をもたないことを強調する場合に関数と呼びます．

手続き・関数呼び出しの実現方法

ここは大事！

- 計算機（CPU）は，手続きを次のように機械語で実行する
  - 手続き呼び出しスタックを用いる（局所変数，引数）.

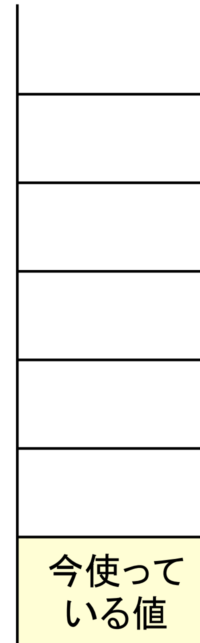
```
#include<stdio.h>
void main() {
  int x;
  int y;
  int answer;
  x = 100;
  y = 50;
  answer = sum(x,y);
  printf("X+Y=%d
\n",answer);
}

int sum(int x0, int y0) {
  return x0 + y0;
}
```

手続き呼び出し

手続き宣言

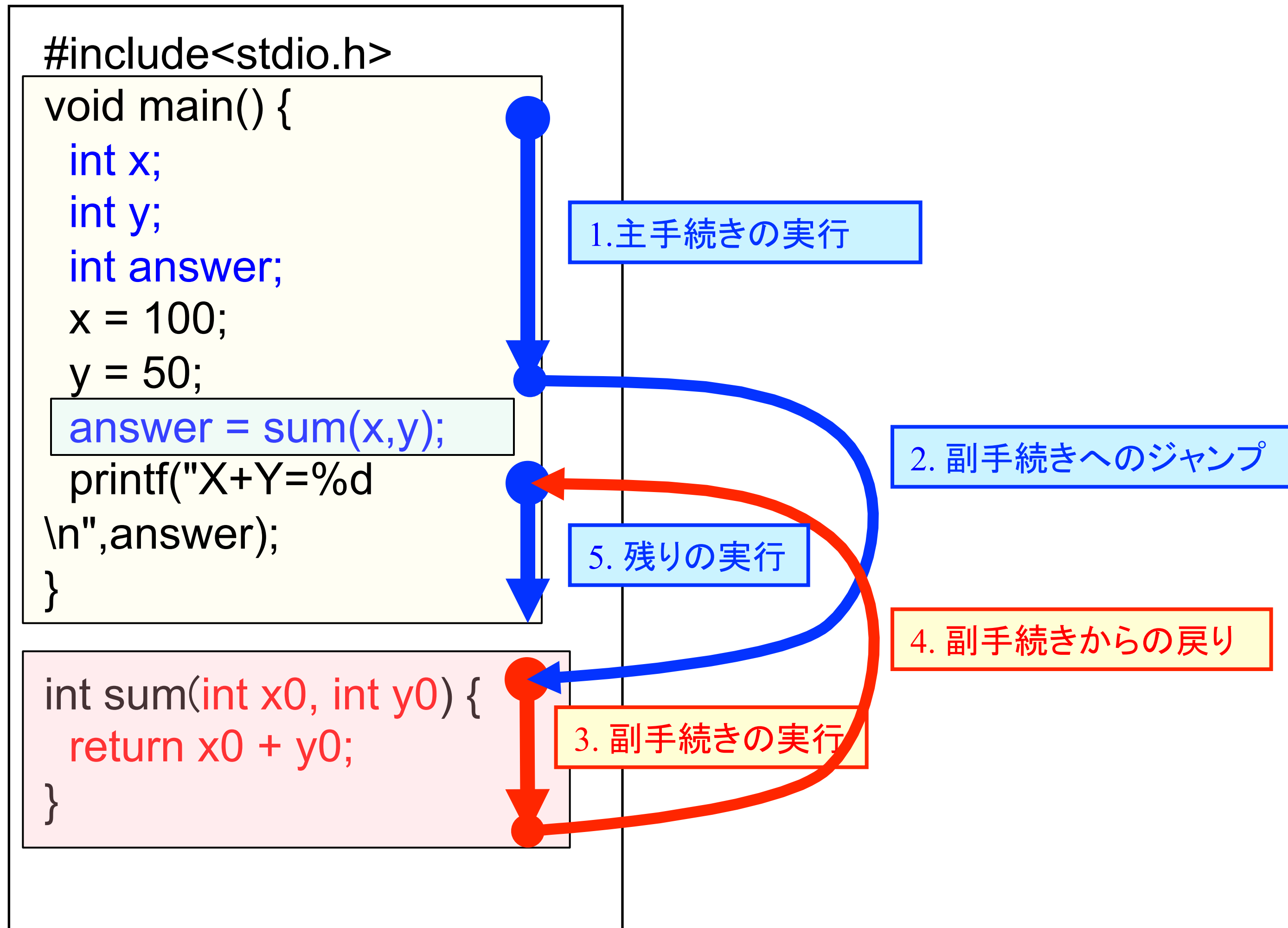
手続き
呼び出し
スタック



発展的内容

手続き・関数呼び出しの実現方法

- CPUによるプログラム（機械語）の実行の流れ.



発展的内容

手続き･関数呼び出しの実現方法

- 計算木は，手続きを次のように機械語で実行する
  - 手続き呼び出しスタックを用いる（局所変数，引数）.

```
#include<stdio.h>
void main() {
  int x;
  int y;
  int answer;
  x = 100;
  y = 50;
  answer = sum(x,y);
  printf("X+Y=%d\n", answer);
}

int sum(int x0, int y0) {
  return x0 + y0;
}
```

手続き呼び出し

手続き宣言

呼び出し側

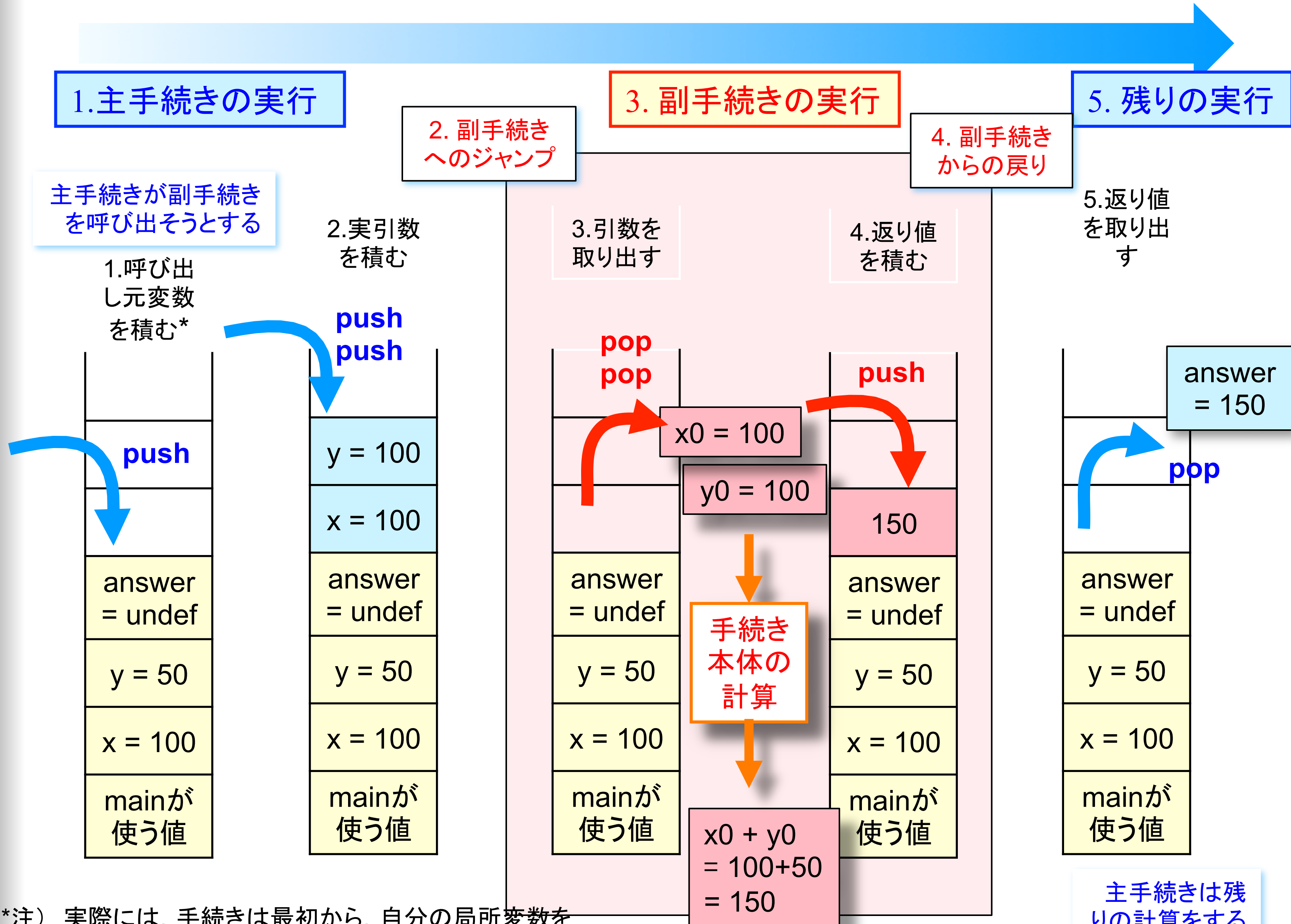
- 呼び出し(main)側の局所変数x, y, answerをスタックに退避する.
- 手続きの引数 xとyを，呼び出しスタックに積む
- 手続き本体のコードを呼びだす

手続き側

- 手続きは呼び出されると，スタックの頂上（トップ）から，実引数(100と50)を取り出し，仮引数xとyとする（数と型はわかる）
- 本体を実行する.
- 計算結果（返り値150=100+50）をスタックのトップに積み，呼び出し側へ戻る.
- 呼び出しに戻ったら，スタックのトップから返り値を取り出し，計算を続ける

手続き・関数呼び出しスタックの動き

発展的内容



*注） 実際には，手続きは最初から，自分の局所変数をスタック上にとっていることが多い（＝スタック変数）

ここは大事！

試験に出ます

2種類の手続き呼び出し(改)

C言語の手続き(関数)呼び出し

- 値による手続き呼び出し(call by value)
  - 引数として手続きに，変数の値を渡す
  - 引数に関数や演算(例:2+3)などが書かれていたときは，それを評価して値(例:5)にしてから渡す.
  - 呼び出し側の変数は変化しない.
- 参照による手続き呼び出し(call by reference)
  - 引数として手続きに，変数の場所(ポインタ・番地)を渡す
  - 呼び出し側に値を引数で戻せる.
- 手続き(関数)の副作用
  - 値による手続き呼び出しでは、引数への副作用はない. 変数に対してどんな操作をしても、その影響は、手続き中で閉じている.
  - 参照による手続き呼び出しを行うと、副作用がある. 手続きを呼び出した後に、変数の内容が変更されている可能性がある

手続きにおける引数の取り扱い

- 値による関数呼び出し(call by value)
  - 引数にとった変数の内容をコピーして利用

```
#include<stdio.h>
void main() {
    int x;
    int y;
    int answer;
    x = 100;
    y = 50;
    answer = sum(x,y);
    printf("X+Y=%d\n",answer);
}

int sum(int x, int y) {
    return x + y;
}
```

演習：手続きによる変数の操作

演習問題

試験に出ます

- 変数の中身を操作する手続き

誤ったプログラム

```
#include<stdio.h>
void main() {
  int x;
  int y;
  x = 100;
  y = 50;
  swap(x,y);
  printf("X=%d, Y = %d\n",x,y);
}
void swap(int x, int y) {
  int tmp;
  tmp = x;
  x = y;
  y = tmp;
  printf("X=%d, Y = %d\n",x,y);
}
```

クイズ：
どこが間違っている
か考えてみよう！

→

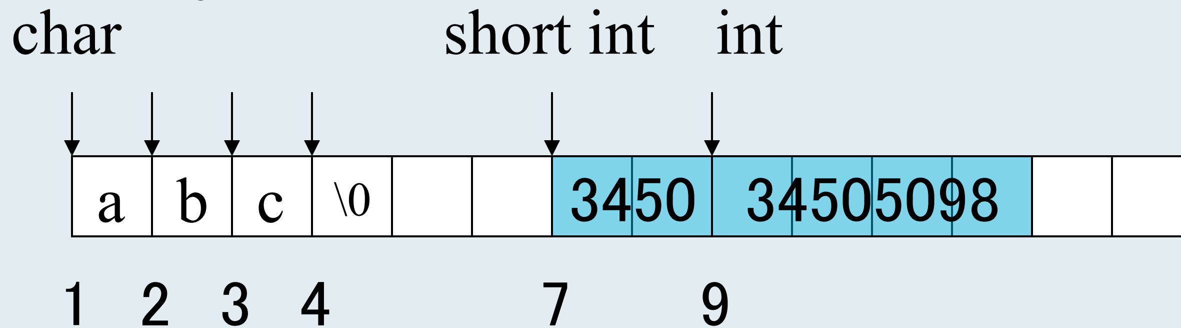
X=50, Y=100
X=100, Y=50

補足．プログラムの説明：二つの変数xとyが保持する値を交換する手続きswap(x, y) を書きたい．応用として，手続きswapは，クイックソート等で用いられる．

このままでは、手続きによって、変数の値を変えられない

ポインタの利用

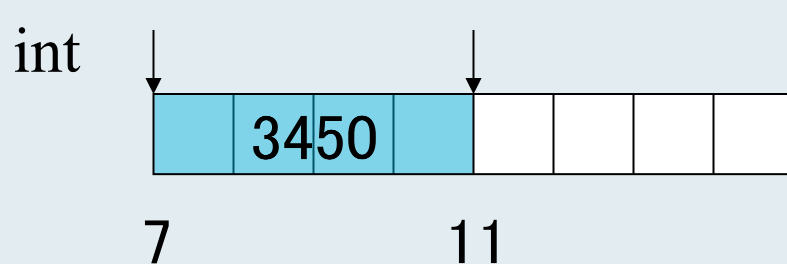
- ポインタ：データを格納しているアドレスの先頭
  - データの種類に応じて、何バイトかの領域が確保されている。



- ポインタの表記
  - &x：　変数xのポインタ
  - *x：　ポインタxに格納されているデータ

ポインタ

- ポインタ変数の表記
  - int a;
    - a : aに対応する領域に格納されている値
    - &a : aのポインタ（aを格納しているメモリのアドレス）
  - int *a;
    - a : aに格納されている値（アドレス）
    - *a : aのアドレスに格納されている値
    - &a : aのポインタ（aを格納しているメモリのアドレス）



int a;　a=3450 &a = 7

int *a;　a=7 *a = 3450

ポインタ変数に関する間違い

- ２つのプログラム

```
int a;
scanf("%d", &a);
```

間違い

```
int *a;
scanf("%d", &a);
```

正解

```
int *a;
scanf("%d", a);
```

↓

scanfにより、アドレスを書き換える
→許されてないメモリ領域へのアクセス
→強制終了

ポインタ変数はアドレス（整数）なので、コンパイラは警告を出すだけで実行可能なファイルを作成

演習:ポインタによる変数の操作(続)

演習問題

試験に出ます

- 先ほどのプログラムの修正

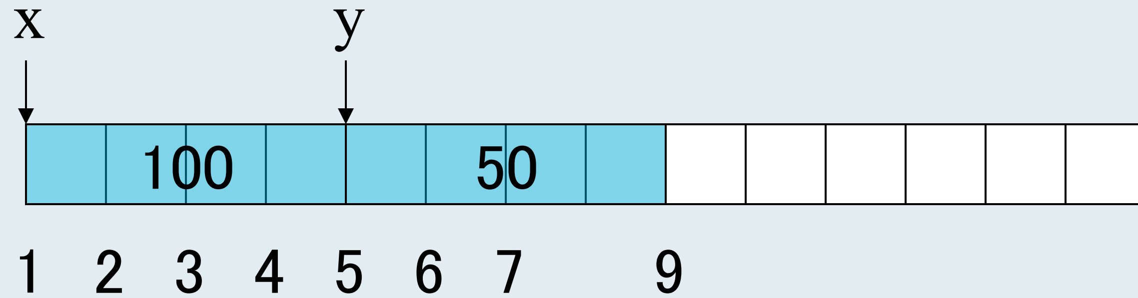
正しいプログラム

```
#include<stdio.h>
int main() {
  int x;
  int y;
  x = 100;
  y = 50;
  swap(&x,&y);                    /* 値ではなくポインタが引数 */
  printf("X=%d, Y = %d\n",x,y);
}
int swap(int *x, int *y) {
  int tmp;
  tmp = *x;                       /* ポインタに対応する */
  *x = *y;                        /* データを操作 */
  *y = tmp;
  printf("X=%d, Y = %d\n",*x,*y);
}
```

プログラムの実行過程(call by value)

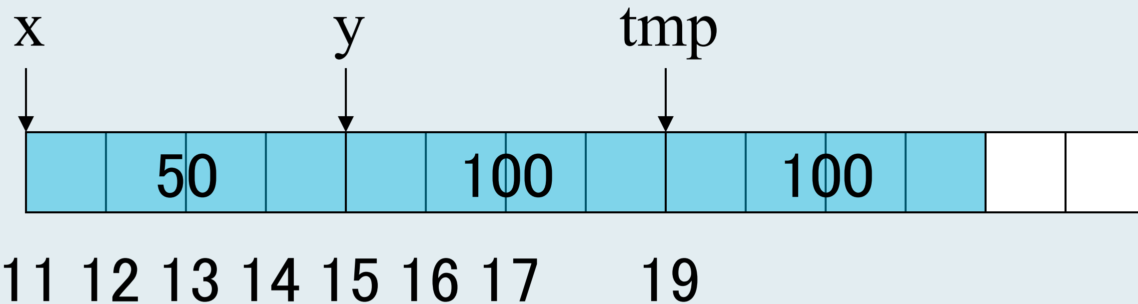
int x = 100;

int y = 50;



x = 100;

y = 50;



プログラムの実行過程(call by reference)

int x = 100;

int y = 50;

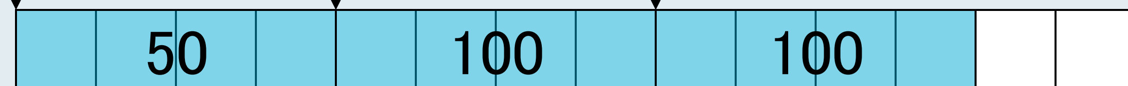
x y

1 2 3 4 5 6 7 9

&x = 1;

& y = 5;

x y tmp



1 2 3 4 5 6 7 9

C言語におけるメモリ空間の利用方法

- コンピュータ内でのメモリ管理
  - 多くの場合：1次元のアドレスでメモリを管理
  - 配列のメモリを確保するためには、未使用の連続したメモリ空間が必要
- 暗黙のメモリ確保と明示的なメモリ確保
  - 暗黙のメモリ確保
    - 変数が宣言される場合には、変数の内容を保存するためのメモリが確保される
    - メモリが確保できない場合は、プログラムは異常終了する
  - 明示的なメモリ確保
    - 必要なメモリの量を宣言し、メモリ空間を関数により確保した後に、変数に割り当てる
    - メモリが確保できない場合には、確保に失敗したという関数の返り値が返る。

- 補足（有村）：スタックメモリとヒープメモリ

明示的なメモリ確保の方法

- メモリ確保の関数：void *malloc(int size)
  - size分だけの連続したメモリを確保し、その先頭のアドレスをvoid型へのポインタとして返す。
- メモリ領域の開放： void free(void *ptr)
  - *ptrで確保されたメモリ空間を解放する（他の目的で利用可能になる）
- 実際の利用方法

```
int *x;                                  /* 配列の先頭を表すポインタ */
x = (int*) malloc(sizeof(int) * 300);    /* 300個のintのためのメモリ領域確保 */
if (x == NULL) {
  printf(“メモリ確保に失敗しました¥n");
}
*x = 2;                                  /* ポインタによる操作 */
printf(“%d\n“, x[0]);                    /* 配列と同じように扱える */
……
```

注意:メモリの確保と変数の初期化

- メモリ確保をしても、変数の値が初期化されるわけではない
  - 以前に使っていた情報が情報として残っている場合がある。
  - mallocには、変数の初期化を行う類似関数void *calloc(int size, int dataSize)がある。

```
#include<stdio.h>
main(){
  int *x;
  int *org;
  printf("Address=%u, x[0]=%d\n", x, *x);
  x = (int*) malloc(100*sizeof(int));
  org = x;
  x[0]=100;
  printf("Address=%u, x[0]=%d\n", x, x[0]);
  free(org);
  x = (int*) malloc(100*sizeof(int));
  printf("Address=%u, x[0]=%d\n", x, x[0]);
}
結果:
Address=3221223944, x[0]=-1073743320
Address=134518456, x[0]=100
Address=134518456, x[0]=100
```

```
#include<stdio.h>
main(){
  int *x;
  int *org;
  printf("Address=%u, x[0]=%d\n", x, *x);
  x = (int*) malloc(100*sizeof(int));
  org = x;
  x[0]=100;
  printf("Address=%u, x[0]=%d\n", x, x[0]);
  free(org);
x = (int*) calloc(100, sizeof(int));
  printf("Address=%u, x[0]=%d\n", x, x[0]);
}結果:

Address=3221223944, x[0]=-1073743320
Address=134518512, x[0]=100
Address=134518512, x[0]=0
```

注意:ポインタによる配列の範囲外へのアクセス

ちょっと高度！

- 各々のポインタ変数には、確保したメモリ空間に関する情報がない。
  - 100個分のメモリを確保した際に、200個目の要素へアクセスができる
    - 他の変数の値を書き換えてしまう可能性がある。
  - Buffer overflowと呼ばれるタイプのエラーの原因となる
    - プログラムの挙動は、処理系によって大きく異なる。

```
#include<stdio.h>
main(){
  int *x, *y;
  int diff;
  x = (int*) malloc(100*sizeof(int));
  y = (int*) malloc(100*sizeof(int));
  diff = y - x;
  *(x+diff+10) = 300;
  printf("Address=%u, x[%d]=%d\n", x+diff+10, diff+10, *(x+diff+10));
  printf("Address=%u, y[%d]=%d\n", y+10, 10, *(y+10));
}
結果:
Address=134518456, x[112]=300
Address=134518456, y[10]=300
```

手続き型言語のその後

- 手続き型言語は，その後，オブジェクト指向言語や，関数型言語と融合して進化した．
  - C++, Java, …
- ほとんどの高級プログラミング言語が，構造化プログラミングの制御構文をもっている．
- また，多くの動的言語（スクリプト言語）が，コンパイル型の処理系を採用しており，プログラミング言語の実装技術においても，基本的である．
- 現在，純粋な手続き型言語は少ないが，（ある意味では）今もプログラミング言語の主流といえる

プログラム言語の歴史

ハードウェア言語
- 機械語、アセンブラ（1940s-1950s）

手続き型言語
- FORTRAN（1960s）
- ALGOL（PASCAL）（1960s）
- 1970s: C言語

オブジェクト志向言語
- 1980s: C++, Objective C, SmallTalk
- 1990s: Java

動的言語・関数型言語
- LISP（1960s）
- Perl, Python, Ruby, Scala ...

本日の授業

- 学ぶこと
  - 手続き型言語とは:
    - 意味のある手続きをひとまとめにした手続きと基本制御構造の組み合わせでプログラムを記述
    - 構造化プログラミングの考え方を反映

- ポイント
  - 変数の有効範囲(スコープ)
  - 手続き・関数の定義
  - [発展] 手続き・関数の実行の仕方
  - 値呼びと参照呼び
  - [発展] ポインタの利用とメモリ確保の方法

まとめ

- 手続き型言語
  - 意味のある手続きをひとまとめにした手続きと基本制御構造の組み合わせでプログラムを記述
  - 構造化プログラミングの考え方を反映
- モジュール化のための方法
  - 手続き･関数の定義
  - 変数のスコープ
  - ポインタの利用
    - メモリ確保の方法