#02
関数
2022 年度 / プログラミング及び実習 III

角川裕次

龍谷大学　先端理工学部

もくじ

今回 (#02) の内容

小テーマ: 関数

第 3 回： 変数の有効範囲と有効期間
第 4 回： 静的/自動変数のメモリ配置

重要概念リスト

- 返却値型 `void`
- `extern` 修飾子
- 有効範囲：ブロック有効範囲とファイル有効範囲
- 関数原型宣言 (プロトタイプ宣言)
- ヘッダ (`#include <stdio.h>`など)
- `return` 文
- 型修飾子 const
- 自動有効範囲 (`auto`) と 静的有効範囲 (`static`)

今回の実習・課題 (manaba へ提出)

実習内容と課題内容は講義途中に提示します

(作成したファイル類は manaba に提出)

第 6-2 節　関数の設計

★重要★

□ 値を返さない関数　p.152

List 6-7 (部分)：スター文字 (アスタリスク*) を n 回表示する関数

```
void put_stars(int n)
{
  while (n-- > 0)
    putchar('*');
}
```

動作内容は「表示」なので関数値を返す必要がない

関数値を返さない関数

返却値型 void を指定

関数の実行を終了する方法：2 通り

- 関数の末尾に実行が到達する場合
- return 文を実行する場合 (返り値は無指定)

左下直角二等辺三角形の表示

List 6-7 (部分)：左下直角二等辺三角形の表示

```
int main(void)
{
  int len;
  printf("左下直角二等辺三角形を作ります。\n");
  printf("短辺：");
  scanf("%d", &len);
  for (int i = 1; i <= len; i++) {
    put_stars(i);
    putchar('\n');
  }
  return 0;
}
```

4 を入力した時の出力

```
*
**
***
****
```

□ 関数の汎用性 p.152

List 6-8：指定の文字 ch を n 回表示する関数

```
void put_chars(int ch, int n)
{
  while (n-- > 0)
    putchar(ch);
}
```

任意の文字を指定できる: 汎用性あり

- 逆に汎用性が凄すぎると使いにくい場合もある
- 汎用性と特殊性 (専用性) の上手な使い分けが大切

put_stars の別実装：汎用的な put_chars を特性化

```
void put_stars(int n)
{
  put_chars('*', int n)
}
```

- n 回表示する制御ロジックを再び書かなくてよい

□ 仮引数を受け取らない関数 p.154

★重要★

List 6-9 (部分)：正の整数値を読み込む

```
int scan_pint(void)
{
  int tmp;
  do {
    printf("正の整数を入力せよ：");
    scanf("%d", &tmp);
    if (tmp <= 0)
      puts("\a正でない数を入力しないでください。");
  } while (tmp <= 0);
  return tmp;
}
```

引数を持たない関数

仮引数並びに void を書く
呼出し：実引数を与えない

呼出しの例

```
  int nx = scan_pint();
```

□ ブロック有効範囲 p.155

★重要★

ブロック有効範囲 (block scope)

ブロック (関数など) 中で宣言された変数：
そのブロックを通用範囲とする (ブロック終端の}まで)

名前が同じでも別ブロックでの宣言なら別物

```
int scan_pint(void)
{
  int tmp;
    ... 略 ...
}

int rev_int(int num)
{
  int tmp = 0;
    ... 略 ...
}
```

有効範囲 (scpoe)：変数が有効な範囲を規定する規則

- 有効範囲の規則はプログラミング言語ごとに異なる

入れ子ブロックでの有効範囲

ブロックの入れ子関係で外側の宣言の内で最も内側の宣言が有効

```
void foo(void)
{
  int tmp;   /*宣言1*/
  {
    int tmp; /*宣言2*/
    for (...) {
      int tmp; /*宣言3*/
      ...    /* ここで使用するtmpは宣言3のもの  */
    }
    ...    /* ここで使用するtmpは宣言2のもの  */
  }
  ...    /* ここで使用するtmpは宣言1のもの  */
  {
    int tmp; /*宣言4*/
    ...    /* ここで使用するtmpは宣言4のもの  */
  }
}
```

□ ファイル有効範囲 p.156

★重要★

ファイル有効範囲 (file scope)

関数の外で宣言された変数：
　　ファイル全体で有効

List 6-10 (部分)

```
#define NUMBER  5
int tensu[NUMBER];

int main(void)
{
    extern int tensu[];  // 省略可
    ...略...    (配列変数tensu[]を使用可能)
}

int top(void)
{
    extern int tensu[];  // 省略可
    ...略...    (配列変数tensu[]を使用可能)
}
```

□ 宣言と定義 p.156

★重要★

定義 (definition) でもある宣言

- 実体を作り出すための宣言
- どのように使われるべきかが定義により明らかになる
- 例: `int tensu[NUMBER];`
  - 配列の実体を確保
  - 配列であること, 要素のデータ型, 要素数が明らか

定義ではない, 単なる宣言

- 実体を使うための宣言
- どのように使われるべきかだけを示す
- 実体は作り出さない
- 例: `extern int tensu[];`
  - 配列の実体は確保しない
  - 配列であること, 要素のデータ型, を示す
  - 配列を使う方法が示されている

extern 修飾子

extern 修飾子

他所で定義が行われていることを表す修飾子

- 型の明示だけを行う
- 定義はこのファイル中の関数の外, または他ファイル中の関数の外で

List 6-10 (部分) : 変数の定義と宣言を同時に行う

```
int tensu[NUMBER]; /* 定義と宣言 */
      ... 以降で配列変数tensu[]を使用可能
```

- 変数の実体を確保 (定義)

List 6-10 (部分) : 変数の宣言だけを行う

```
int top(void)
{
  extern int tensu[]; /* 宣言 */
      ... 以降(関数内)で配列変数tensu[]を使用可能
}
```

□ 関数原型宣言 p.157

★重要★

関数原型宣言 (プロトタイプ)

関数の返却値型や仮引数の型の並びの宣言

目的：関数呼び出しの適切さ (返却値型や仮引数の型) の検査

- ソースコード上で関数呼び出しが現れた時点でチェックしたい
- 言語処理系はソースコードの先頭から読み進めるため
- 検査にはその時点で関数の返却値型や仮引数の型の並びの情報が必要

解決法：関数呼出しの前方で関数原型宣言を行う

```
int top(void);    /* 関数原型宣言 */

int main(void)
{    ... 略 ...
  printf("最高点=%d\n", top());   /* 呼出し */
     ... 略 ...    }

int top(void) /* 関数定義 */
{    ... 略 ...    }
```

関数原型宣言の例

関数定義

- 例: `int top(void) { ... 略 ... }`
- 関数を定義
- どう関数が使われるべきかも分かる

関数原型宣言

- 例: `int top(void);`
- 関数は定義しない
- 関数がどう使われるべきかだけが示されている

ソースコード中で関数を書き並べる順序 (1/2)

(教科書とは逆のことを書きます; 主観です)

目的：500 行のソースコードを読解可能な書き方をするには

推奨：
・呼び出す側の関数 (main 関数など) を先に書く
・呼び出される側の関数を後に書く

理由：
・全体から詳細へと段階的に読解ができる
・呼び出す側の関数：プログラム構造の大枠
・呼び出される側の関数：事細かなこと

ソースコード中で関数を書き並べる順序 (2/2)

(教科書とは逆のことを書きます; 主観です)

教科書で推奨している方法：呼び出す関数を後に書く
・コードの詳細部分から読まされる

300 行以上のプログラム読解は困難
・実用的なプログラム開発には役立たない
・私はこの方法はすすめないです

クイズの「これ何の写真?」
- 倍率 100 倍の拡大写真を見せる → 50 倍 → 20 倍 → 5 倍 → 1 倍
- 最後まで何か分からない：だからこそクイズとして成立
- 逆順：一発で全体が分かる

□ ヘッダとインクルード p.158

Q. いつもソースコード先頭にある `#include <stdio.h>` って一体何?

A. ヘッダ (header) をインクルードしています

ライブラリ関数の関数原型宣言：記述済みのファイルを予め用意

- ヘッダと呼ばれる
- printf, putchar,... ： ファイル stdio.h
- isspace, isalpha,... ： ファイル ctype.h
- sin, cos, ... ： ファイル math.h

`#include` 指令でソースコード中に取り込んで利用

- 例: `#include <stdio.h>`
- 指定のファイルの中身で include 指令を置き換え
- インクルード (include) とも呼ぶ

ヘッダのファイルはどこにある? (気にしなくて良い)

(Ubuntu 18.04 など Unix 系 OS の場合)

`#include <...>` で取り込むファイル:
ディレクトリ`/usr/include` の下

- `/usr/include/stdio.h`
- `/usr/include/ctype.h`
- `/usr/include/math.h`

  ...

`#include "..."` で取り込むファイル:
ソースコードのディレクトリの下

ヘッダをインクルードする仕組

C プリプロセッサ (通称 cpp) の機能

コンパイル前に cpp が呼び出される

- 入力：C 言語のソースファイル
- 動作：ファイル中の `#include` 文をヘッダの内容に置き換え
- 出力：(等価な) C 言語のソースファイル

cpp が出力したソースファイルを C コンパイラ本体がコンパイル

他の cpp の機能

- マクロ定義 `#define` 文とマクロ展開
- 条件コンパイル `#ifdef` 文
- 条件コンパイル `#if`,`#else`,`#endif` 文

□ 関数の汎用性 p.159

List 6-10 (部分) : 関数 top

```
int top(void)
{
  extern int tensu[]; /* 配列の宣言（省略可）*/
  int max = tensu[0];

  for (int i = 1; i < NUMBER; i++)
    if (tensu[i] > max)
      max = tensu[i];
  return max;
}
```

これでは汎用性に欠ける

- 配列 tensu と 要素数 NUMBER に決め打ち

対象の配列と要素数に自由度があると利用範囲が増える

- 引数で渡せるようにするとよい

□ 配列の受け渡し　p.160

List 6-11 (部分)：配列を引数とするプログラム例

```
int max_of(int v[], int n)
{
    ... 略 ...
}

int main(void)
{
  int eng[NUMBER];    /* 英語の点数 */
  int mat[NUMBER];    /* 数学の点数 */
  int max_e, max_m;   /* 最高点 */
    ... 略 ...
  max_e = max_of(eng, NUMBER); /* 英語の最高点 */
  max_m = max_of(mat, NUMBER); /* 数学の最高点 */
  return 0;
}
```

仮引数 `int v[]`：引数は整数配列を表す

引数の配列の使用

List 6-11 (部分)：要素数 n の配列 v の最大値を返す

```
int max_of(int v[], int n)
{
  int max = v[0];
  for (int i = 1; i < n; i++)
    if (v[i] > max)
      max = v[i];
  return max;
}
```

top よりも汎用性が高い

- 対象の配列と要素数を引数で指定できる
- 他の配列での最大値を求めるのに使える

関数に渡された配列

List 6-11 (部分)：要素数 n の配列 v の最大値を返す

```
int max_of(int v[], int n)
{
    ... 略 ...
}
```

仮引数に与えられる配列：呼び出し側の実引数の配列そのもの

- 関数内で配列の要素の値を変更 ⇒ 呼び出し元の配列の値も変化

・main 関数での呼出 max_of(eng, NUMBER)：
  max_of() の仮引数 v[] は配列 eng[] に
- v[0] は eng[0] と同一 (同じメモリアドレス)
- v[1], v[2], ... も同様

・main 関数での呼出 max_of(mat, NUMBER)：
  max_of() の仮引数 v[] は配列 mat[] に
- v[0] は mat[0] と同一 (同じメモリアドレス)
- v[1], v[2], ... も同様

★重要★

□ 配列の受け渡しと const 型修飾子 p.162

const 型修飾子

値の変更禁止をソースコード上で明示する方法
関数内で代入をするソースコードをコンパイルエラーにする

`const` なし：配列の内容を呼び出した関数内で書き換えできる

- 関数内で配列の要素の値を変更 $\Rightarrow$ 呼び出し元の配列の値も変化

配列の内容を変更されては呼び出し元が困る場合もあり

- 知らずに関数内部で書き換えてしまうコードを書くかも...
- 解決法：`const` 型修飾子の導入

const 型修飾子の使用例

List 6-12 (部分)：配列の内容を表示する関数

```
void print_array(const int v[], int n)
{
  printf("{ ");
  for (int i = 0; i < n; i++)
    printf("%d ", v[i]);
  printf("}");
}
```

仮引数 `const int v[]`

- 関数内での値の変更を禁止
- 変更 (代入) するプログラムコードを書けばコンパイルエラー

関数 `print_array` を呼び出す側

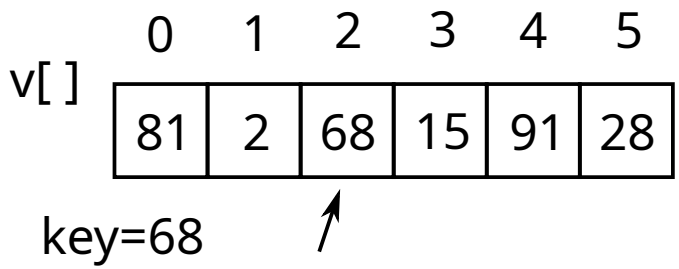
- 配列の内容が変更されない保証を得る

□ 線形探索 (逐次探索) p.164

関数 search： 配列 v 中の値 key を探索して一致する添字を返す

返す値 i の条件： v[i] = key が成立

- 存在しなければ −1 を返す



どうやって実現する?

- i=0,1,2,... としながら
- もし v[i] = key なら i の値をリターン
- ただし配列の終わりに来る (見つからない) と終了: −1 を返す

線形探索のコード例

List 6-13 (部分)：配列 v 中の値 key を探索して添字を返す

```
#define FAILED     -1    /* 探索失敗 */

int search(const int v[], int key, int n)
{
  int i = 0;
  while (1) {
    if (i == n)
      return FAILED;      /* 探索失敗 */
    if (v[i] == key)
      return i;           /* 探索成功 */
    i++;
  }
}
```

呼出

List 6-13 (部分) : 線形探索の main 関数

```
#define NUMBER     5     /* 要素数 */

int main(void)
{
  int ky, idx;
  int vx[NUMBER];

        ... 略(配列の値と探索する値を入力) ...

  idx = search(vx, ky, NUMBER); /*配列vxからkyを探索*/
  if (idx == FAILED)
    puts("\a探索に失敗しました。");
  else
    printf("%dは%d番目にあります。\n", ky, idx + 1);
  return 0;
}
```

□ 番兵法 p.166

探索値の判定とループ終了判定を同時に行うプログラミング技法

- 探索対象の配列 v は要素数を 1 つ余分に確保 (要素数 n+1)
- 配列の最後の要素 v[n] に探索値 key を代入して探索開始

List 6-14：番兵

```
int search(int v[], int key, int n)
{
  int i = 0;
  v[n] = key;  /* 番兵を格納 */
  while (1) {
    if (v[i] == key)
      break;
  }
  return i < n ? i : FAILED;
}
```

実行時間の高速化に有効：比較だけで探索を終了 (ループ終了判定なし)

- 検索値がない: v[n]=key で必ず終了
- 検索値がある: v[i]=key (i < n) で終了

番兵法

List 6-14：配列宣言に注意 (番兵用の要素を確保)

```
int main(void)
{
  int ky, idx;
  int vx[NUMBER + 1]; //★番兵用の要素を確保★
      ... 略(配列の値と探索する値を入力) ...
  if ((idx = search(vx, ky, NUMBER)) == FAILED)
    puts("\a探索に失敗しました。");
  else
    printf("%dは%d番目にあります。\n", ky, idx + 1);
  return 0;
}
```

ひとつ多く要素を確保

- vx[0]
- vx[1]

  ...
- vx[NUMBER-1]　　vx[0] からここまでを入力値の記憶に使用
- vx[NUMBER]　　ここが 1 つ多い... 番兵用に使用

if 文の条件式の解説

if 文

```
if ((idx = search(vx, ky, NUMBER)) == FAILED) ...
```

条件式だけに注目:

```
((idx = search(vx, ky, NUMBER)) == FAILED)
```

条件式の構造: `((v = f()) == X)`

条件式の値の計算実行の流れ

1. 関数の呼び出し: `f()`
2. 変数への関数値の代入: `v=...`
3. 代入された値と `X` との比較: `(...) == X`

条件式の値 if 文の条件式の真偽が決定 (0: 偽, 非 0: 真)

注意 (番兵法)

正しく読みやすいプログラムを書けるようなるのが先決

- 高速化の技巧はその後に考えること

バグありプログラムをどんなに高速化してもバグありのまま

分かりにくいプログラムは高速化する途中でバグを仕込んでしまう

- バグ取りきれなくて修羅場なのがおやくそく

□ 多次元配列の受け渡し　p.170

List 6-16 (部分)：多次元配列を関数に渡す

```
int main(void)
{
  int tensu1[4][3] = { {91, 63, 78}, {67, 72, 46},
                       {89, 34, 53}, {32, 54, 34} };
  int tensu2[4][3] = { {97, 67, 82}, {73, 43, 46},
                       {97, 56, 21}, {85, 46, 35} };
  int sum[4][3]; /* 合計 */
  mat_add(tensu1, tensu2, sum); /* 合計を求める */
  puts("1回目の点数"); mat_print(tensu1);
  puts("2回目の点数"); mat_print(tensu2);
  puts("合計点");      mat_print(sum);
  return 0;
}
```

2 次元配列 (4 行 3 列)
関数呼び出し時の実引数：配列名を指定

多次元配列を引数とする関数の例 (1/2)

List 6-16 (部分)：4 行 3 列の行列 a と b の和を c に格納

```
void mat_add(const int a[4][3], const int b[4][3],
             int c[4][3])
{
  for (int i = 0; i < 4; i++)
    for (int j = 0; j < 3; j++)
      c[i][j] = a[i][j] + b[i][j];
}
```

多次元配列を引数とする関数の例 (2/2)

List 6-16 (部分)：4 行 3 列の行列を表示

```
void mat_print(const int m[4][3])
{
  for (int i = 0; i < 4; i++) {
    for (int j = 0; j < 3; j++)
      printf("%4d", m[i][j]);
      putchar('\n');
    }
}
```

多次元配列の仮引数

★重要★

n 次元配列の仮引数の書き方

- $n-1$ 次元目までは要素数の指定が必須
- n 次元は要素数の指定はしなくてもよい

List 6C-1 (部分) : 関数 `mat_print` の仮引数の別の書き方

```
void mat_print(const int m[][3], int n)
{
    ... 略 ...
}
```

(仮引数の場合) `int m[4][3]` を `int m[][3]` と書いても同じ

例: 4 行 3 列の 2 次元配列の仮引数の書き方 (2 通り)

- `int m[][3]` : 1 次元目まで要素数を指定 (2 次元目を省略した場合)
- `int m[4][3]` : 2 次元目も要素数を指定

多次元配列の仮引数 (2)

★重要★

関数 mat_print は以下の呼び出しができる

```
int x[2][3];
int y[4][3];

mat_print(x, 2);
mat_print(y, 4);
```

- 仮引数を m[][3] と宣言しているため

再掲 List 6C-1 (部分) : 関数 mat_print の仮引数の別の書き方

```
void mat_print(const int m[][3], int n)
{
    ... 略 ...
}
```

- もし仮引数の宣言が m[4][3] だと x を引数に出来ない

第 6-3 節　有効範囲と記憶域期間

□ 有効範囲と識別子の可視性 p.172

★重要★

List 6-17：識別子の有効範囲 (scope) の例

```
int x = 75;    /* A : ファイル有効範囲 */

void print_x(void)
{
  printf("x = %d\n", x);          /* → A を参照 */
}

int main(void)
{
  int x = 999; /* B : ブロック有効範囲 */
  print_x();
  printf("x = %d\n", x);          /* → B を参照 */
  for (int i = 0; i < 5; i++) {
    int x = i * 100; /* C : ブロック有効範囲 */
    printf("x = %d\n", x);        /* → C を参照 */
  }
  printf("x = %d\n", x);          /* → B を参照 */
  return 0;
}
```

同名の変数がある場合 (1/2)

★重要★

規則 1: ファイル有効範囲とブロック有効範囲

両方に同名の変数が存在する場合：
　　ブロック有効範囲の変数が有効となる

例：List 6-17 (部分) — 2 つの変数 x

```
int x = 75;    /* A : ファイル有効範囲 */

int main(void)
{
  int x = 999; /* B : ブロック有効範囲 */
  printf("x = %d\n", x);          /* → B を参照 */
  return 0;
}
```

同名の変数がある場合 (2/2)

★重要★

規則 2: 入れ子のブロック有効範囲

両方に同名の変数が存在する場合：
より内側の変数が有効となる

例：List 6-17 (部分) — 2 つの変数 x

```
int main(void)
{
  int x = 999; /* B：ブロック有効範囲 */
  for (int i = 0; i < 5; i++) {
    int x = i * 100; /* C：ブロック有効範囲 */
    printf("x = %d\n", x);        /* → C を参照 */
  }
  return 0;
}
```

□ 記憶域期間 (strage duration)

★重要★

変数がいつ利用可能になり・いつ消滅するかを表す概念

- 変数の生存期間 (寿命)
- 最初から最後まで存在し続けるわけではない
- あるときに生まれ, あるときに消える

C 言語では 2 通り : 自動記憶域期間と静的記憶域期間

自動記憶域期間

振る舞いの直感：以前に関数を実行したときの変数値は失念

自動記憶域期間 (automatic strage duration)

変数の宣言方法：関数 (ブロック) 中で auto(省略可) を付加
・変数の生成：プログラムの流れが宣言を通過するとき
・変数の消滅：プログラムの流れがブロックを抜けるとき
初期値：明示的に与えられなければ未定義

- 例 1: `auto int x;`
- 例 2: `int x;`
- 例 3: `int x = 0;`

注意：初期値は明示的に与えられなければ未定義

- 実行の度に初期値が違う可能性あり
- 性質：前回の値を覚えていない

静的記憶域期間

★重要★

振る舞いの直感：以前に関数を実行したときの変数値を記憶

静的記憶域期間 (static strage duration)

変数の宣言法：関数 (ブロック) 中または関数外で `static` を付加
・変数の生成：プログラムの実行開始時 (main 関数実行前)
・変数の消滅：プログラムの実行終了時
初期値：明示的に与えられない場合は 0

- 例 1: `static int y;`
- 例 2: `static int x = 256;`

注意：変数宣言文を 2 回以上実行する場合

- 実行の度に代入するのではない
- 初期化はプログラム起動時の 1 度だけ
- 性質：前回の値を覚えている

C 言語の変数の記憶域期間： まとめ

自動記憶域期間

- 宣言方法： 関数中で宣言する変数で `auto` 付加 (省略可)
- 記憶域期間： 関数を前回実行したときの変数値は記憶していない

静的記憶域期間 (1)

- 宣言方法： 関数中で宣言する変数で `static` 付加
- 記憶域期間： 関数を前回実行したときの変数値は記憶している

静的記憶域期間 (2)

- 宣言方法： 関数外で宣言する変数
- 記憶域期間： 関数呼び出しに関係なくずっと変数値を記憶する

★重要★

プログラム書いて確認してみる (List 6-18; 一部改変)

- 関数 func 内で各変数値を 1 増加
- 次回呼出し時にその値を覚えているか?

```
#include <stdio.h>

int fx = 0;

void func(void) {
  static int sx = 0;
  int        ax = 0;
  print("%3d %3d %3d\n",
        ax++, sx++, fx++);
}

int main(void){
  int i = 0;
  for (i = 0; i < 5; i++) {
    func();
  }
  return 0;
}
```

実行結果

```
(ax  sx  fx)
  0   0   0
  0   1   1
  0   2   2
  0   3   3
  0   4   4
```

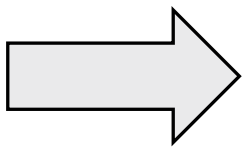
ax : 以前の値は失念
(自動記憶域期間)
sx : 以前の値を記憶
(静的記憶域期間)
fx : 以前の値を記憶
(静的記憶域期間)

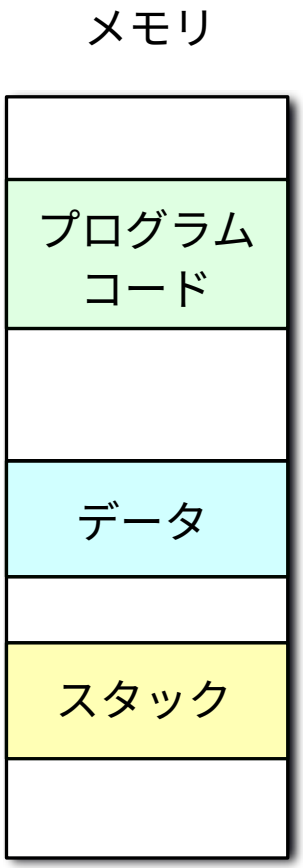
ソースコードからメモリへの配置

プログラムコード	CPU 機械語の列
データ	静的記憶域期間の変数
スタック	自動記憶域期間の変数と関数終了時の復帰情報

```
int fx = 0;

void func(void) {
  static int sx = 0;
  int        ax = 0;
  print("%3d %3d %3d\n",
        ax++, sx++, fx++);
}

int main(void){
  int i = 0;
  for (i = 0; i < 5; i++) {
    func();
  }
  return 0;
}
```

ソースコード



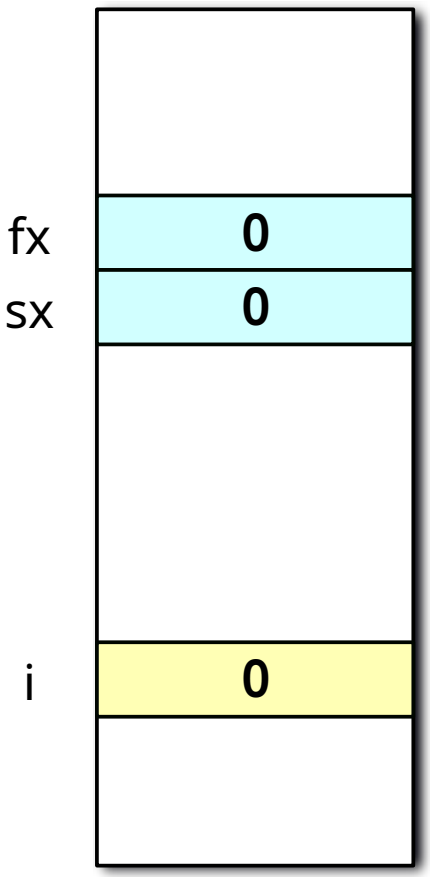


実行時のメモリ変化 (1) — main 呼出直後 ($i = 0$)

変数 ax はまだ存在してない

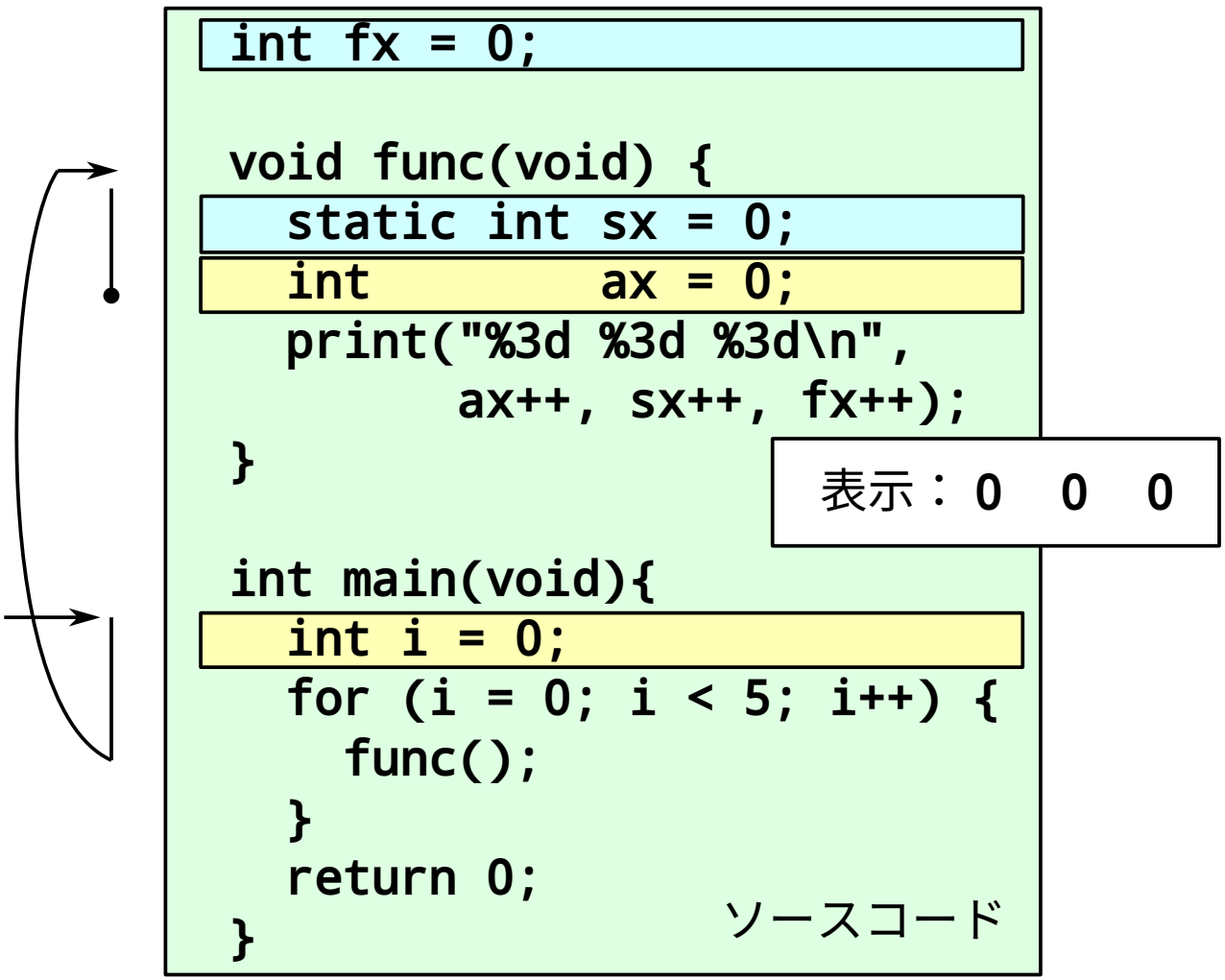
```
int fx = 0;

void func(void) {
  static int sx = 0;
  int        ax = 0;
  print("%3d %3d %3d\n",
        ax++, sx++, fx++);
}

int main(void){
  int i = 0;
  for (i = 0; i < 5; i++) {
    func();
  }
  return 0;
}
```

ソースコード



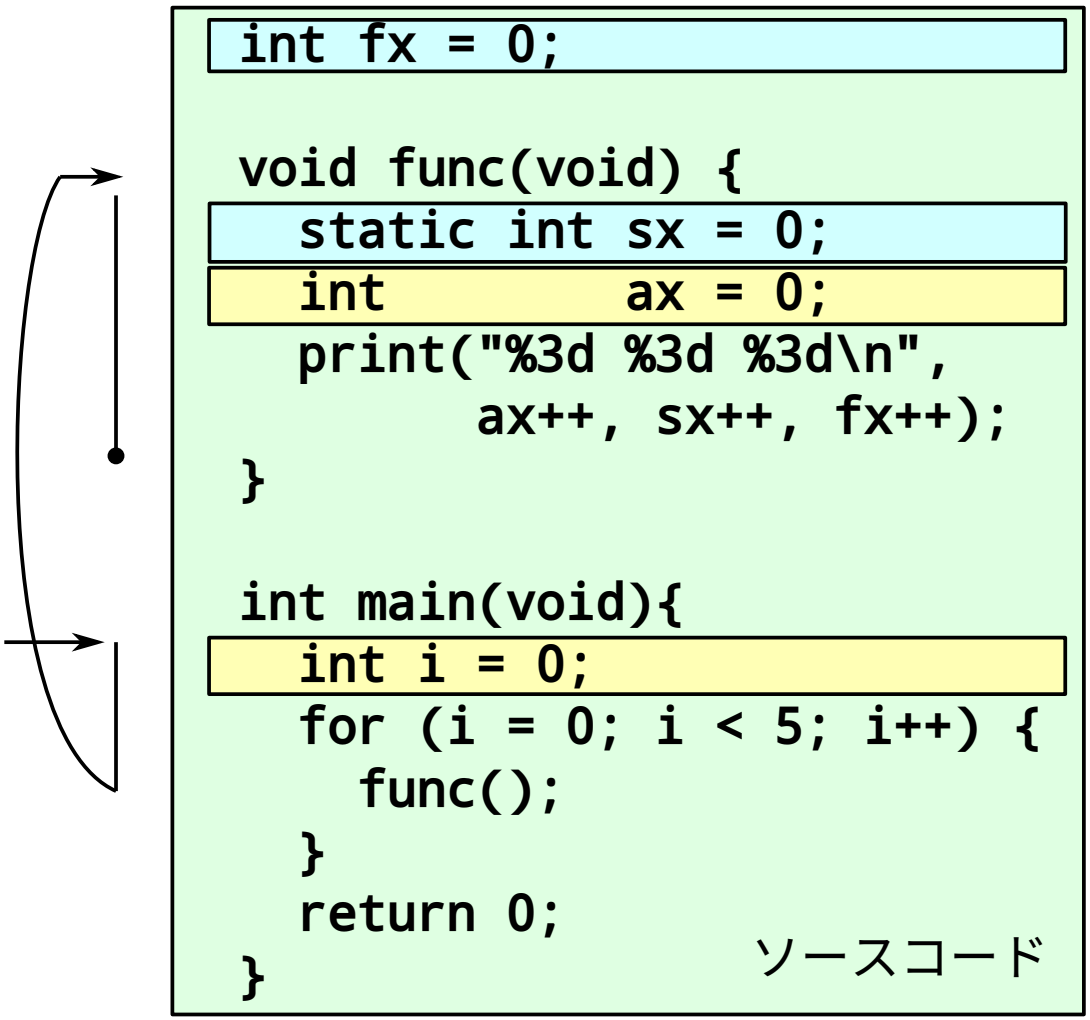
実行時のメモリ変化 (2) — func 呼出直後 ($i = 0$)

変数 ax が関数 func の呼出により発生



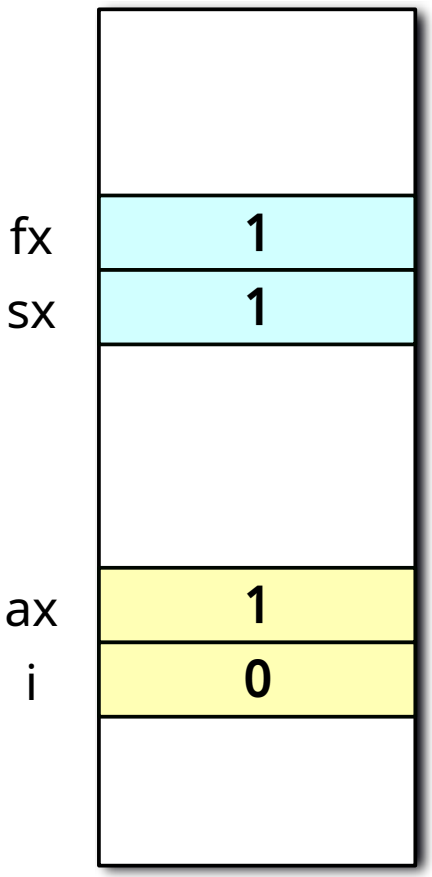
実行時のメモリ変化 (3) — func 終了直前 ($i = 0$)

変数 fx, sx, ax それぞれを 1 増加



```
int fx = 0;

void func(void) {
  static int sx = 0;
  int        ax = 0;
  print("%3d %3d %3d\n",
        ax++, sx++, fx++);
}

int main(void){
  int i = 0;
  for (i = 0; i < 5; i++) {
    func();
  }
  return 0;
}
```

ソースコード

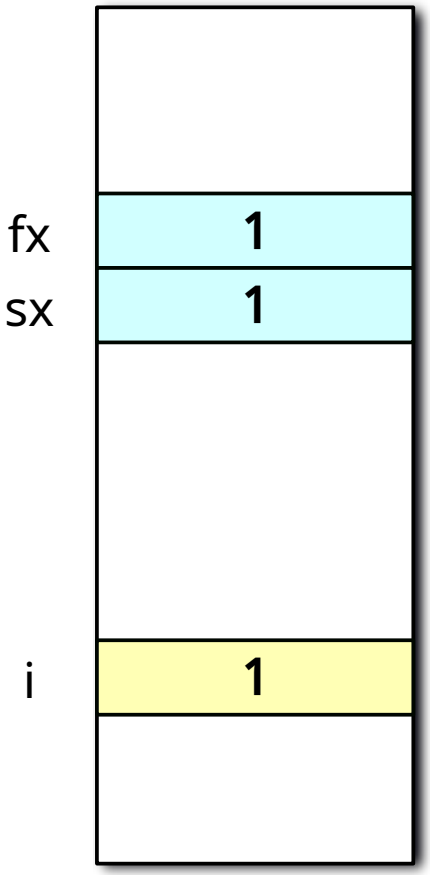


実行時のメモリ変化 (4) — main, ループ 2 回目 ($i = 1$)

変数 ax は関数 func の終了により消滅

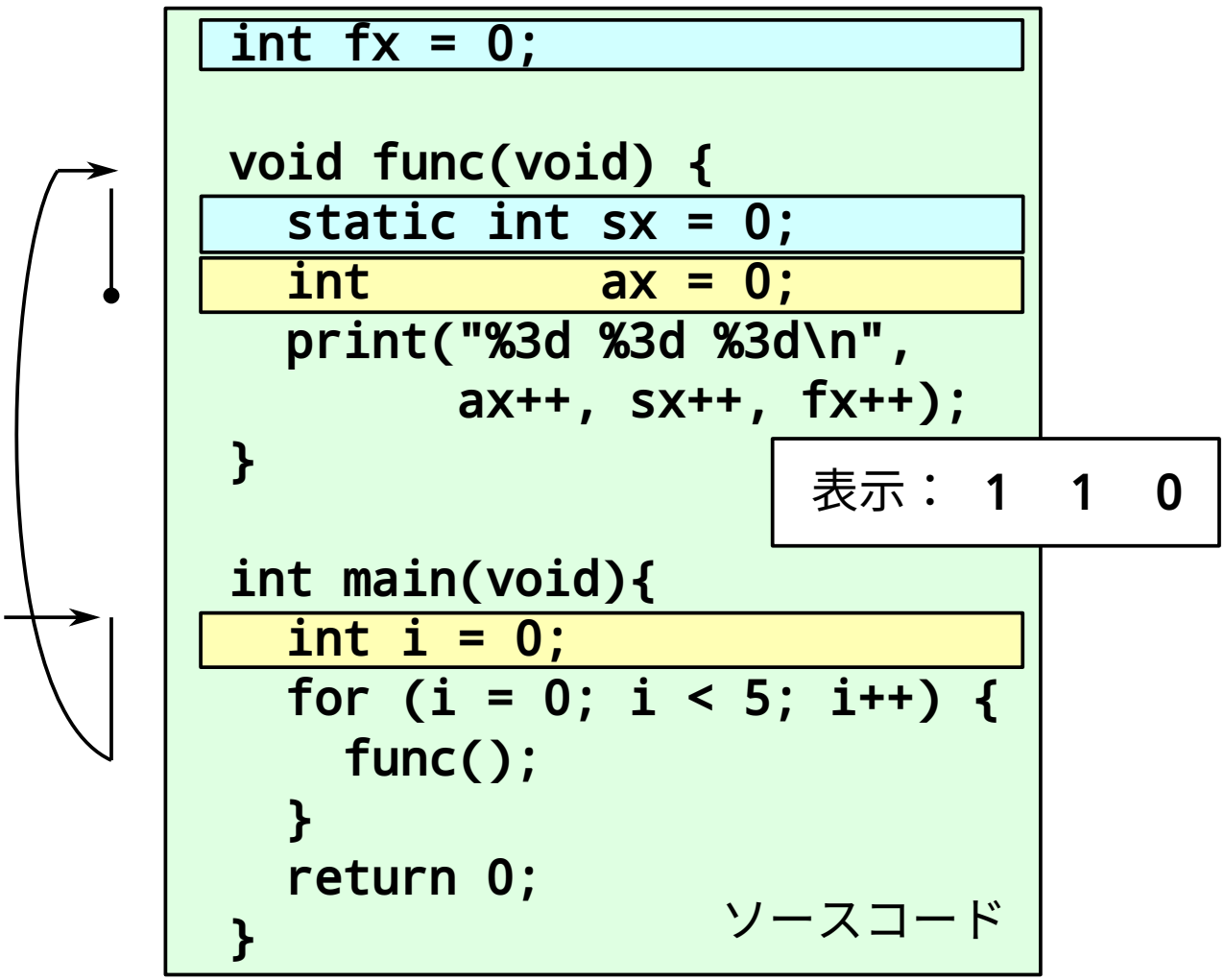
```
int fx = 0;

void func(void) {
  static int sx = 0;
  int        ax = 0;
  print("%3d %3d %3d\n",
        ax++, sx++, fx++);
}

int main(void){
  int i = 0;
  for (i = 0; i < 5; i++) {
    func();
  }
  return 0;
}
```

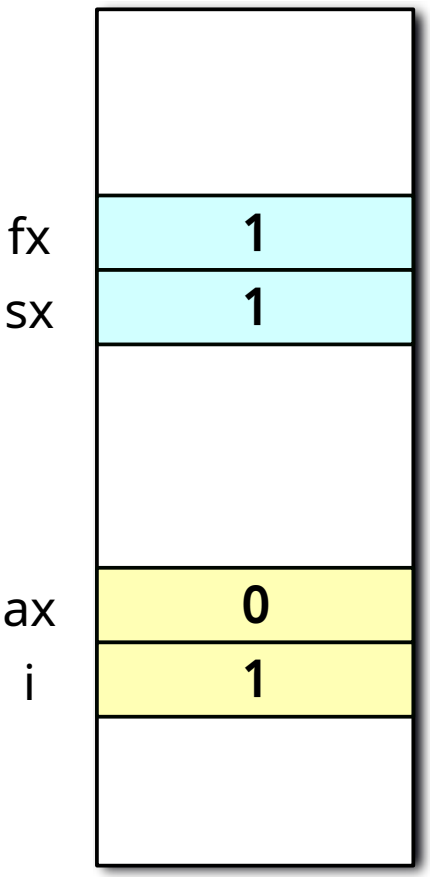
ソースコード



実行時のメモリ変化 (6) — func 呼出直後 ($i = 1$)

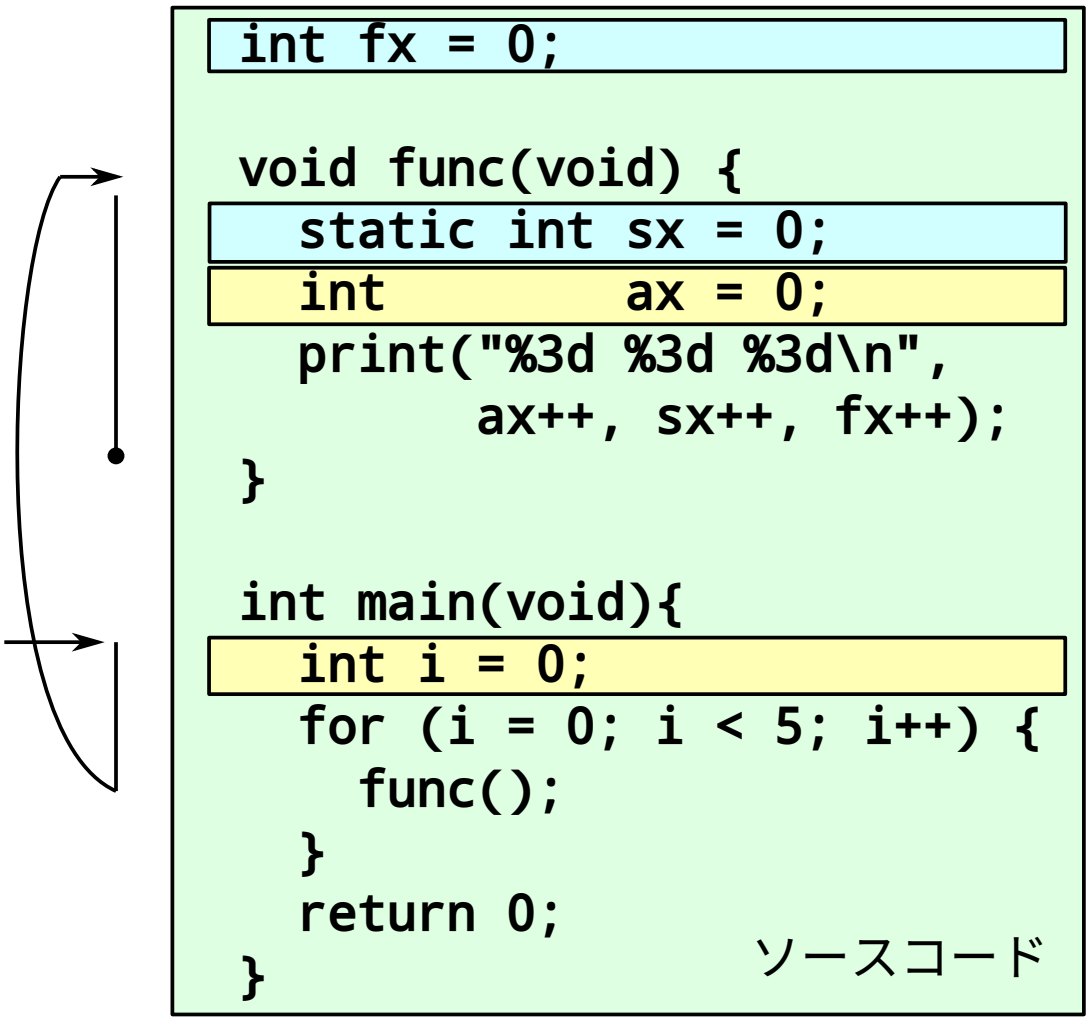
変数 ax が関数 func の呼出により発生：変数値は 0 (指定の初期値) に





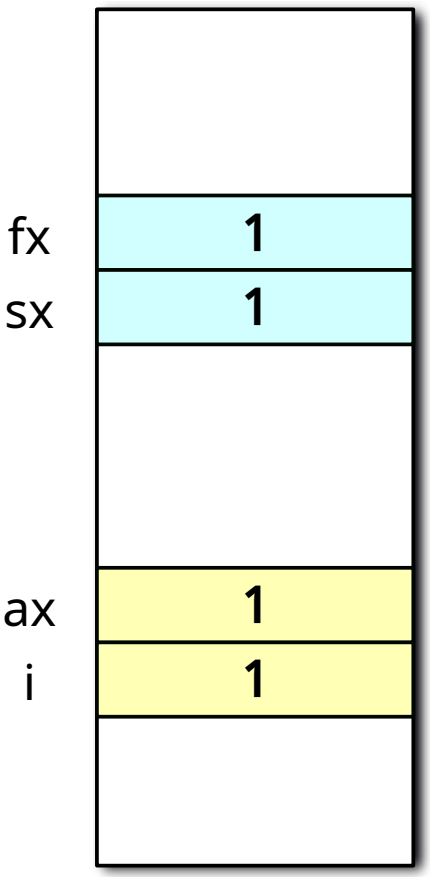
実行時のメモリ変化 (7) — func 終了直前 ($i = 1$)

変数 fx, sx, ax それぞれを 1 増加



```
int fx = 0;

void func(void) {
  static int sx = 0;
  int        ax = 0;
  print("%3d %3d %3d\n",
        ax++, sx++, fx++);
}

int main(void){
  int i = 0;
  for (i = 0; i < 5; i++) {
    func();
  }
  return 0;
}
```

ソースコード



おわり