

## 2.2.有効範囲と記憶クラス

有効範囲（グローバル変数とローカル変数、scope）

記憶クラス（静的変数と自動変数）

# 変数の有効範囲

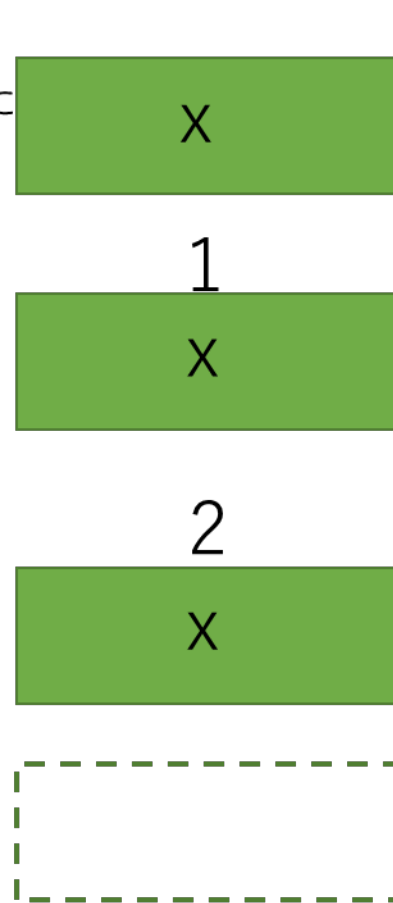
```
int main (void){  
    int x;  
    x=1;  
    x=x+1;  
    return 0;  
}
```

このタイミングで、変数の値を記憶するためのハコがメモリ領域に用意される

このタイミングで、ハコの中に1という値が代入される

このタイミングで、ハコの中にあった1が2に置き換わる

このタイミングで、ハコを廃棄する（メモリを解放する）



※ポイント：変数を記述する場所やタイミングによって、**グローバル変数**と**ローカル変数**に分かれ、それぞれスコープが大きく異なる⇒次の頁参照

この範囲が、一つの変数の存在する寿命  
⇒有効範囲（スコープ:scope）と呼ぶ

# 変数の有効範囲

## List2.1

```
#include <stdio.h>
#pragma warning( disable : 4996 )
int x = 111;

void print_x(void) {
    printf("[print_x] x=%d\n", x);
}

int main(void) {
    printf("x=%d\n", x);
    int i;
    int x = 999;
    print_x();
    printf("[main] x=%d\n", x);

    for (i = 0; i < 5; i++) {
        int x = i;
        printf("[for] x=%d\n", x);
    }
    printf("[main] x=%d\n", x);
    print_x();
}
```

グローバル変数

A

ローカル変数

B

ローカル変数

C

A

x

B

x

C

x

## 実行結果

```
x=111
[print_x] x=111
[main] x=999
[for] x=0
[for] x=1
[for] x=2
[for] x=3
[for] x=4
[main] x=999
[print_x] x=111
```

※ポイント：

**グローバル変数**：プログラムの開始時から終了時までが寿命

**ローカル変数**：そのブロック内（関数やfor, if内）が寿命

※ポイント：同じ名前の変数をローカル変数として定義しても、元の変数は消えない（メモリ領域に別のハコが用意されるだけ）。同じ名前の場合、**ローカル変数が優先的に呼び出される**。

# 変数の有効範囲

## List2.2

```
#include <stdio.h>
#pragma warning( disable : 4996 )
int a;
void kakunin1(int a) {
    a = a;
}
void kakunin2(int b) {
    int a;
    a = b;
}
void kakunin3(int b) {
    a = b;
}
void main() {
    printf("a=%d\n", a);
    a = 10;
    printf("a=%d\n", a);
    kakunin1(20);
    printf("a=%d\n", a);
    kakunin2(30);
    printf("a=%d\n", a);
    kakunin3(40);
    printf("a=%d\n", a);
}
```

### 【問題】

左のコードの出力結果は下記のようなになる。理由を考えよ

### 実行結果

```
a=0
a=10
a=10
a=10
a=40
```

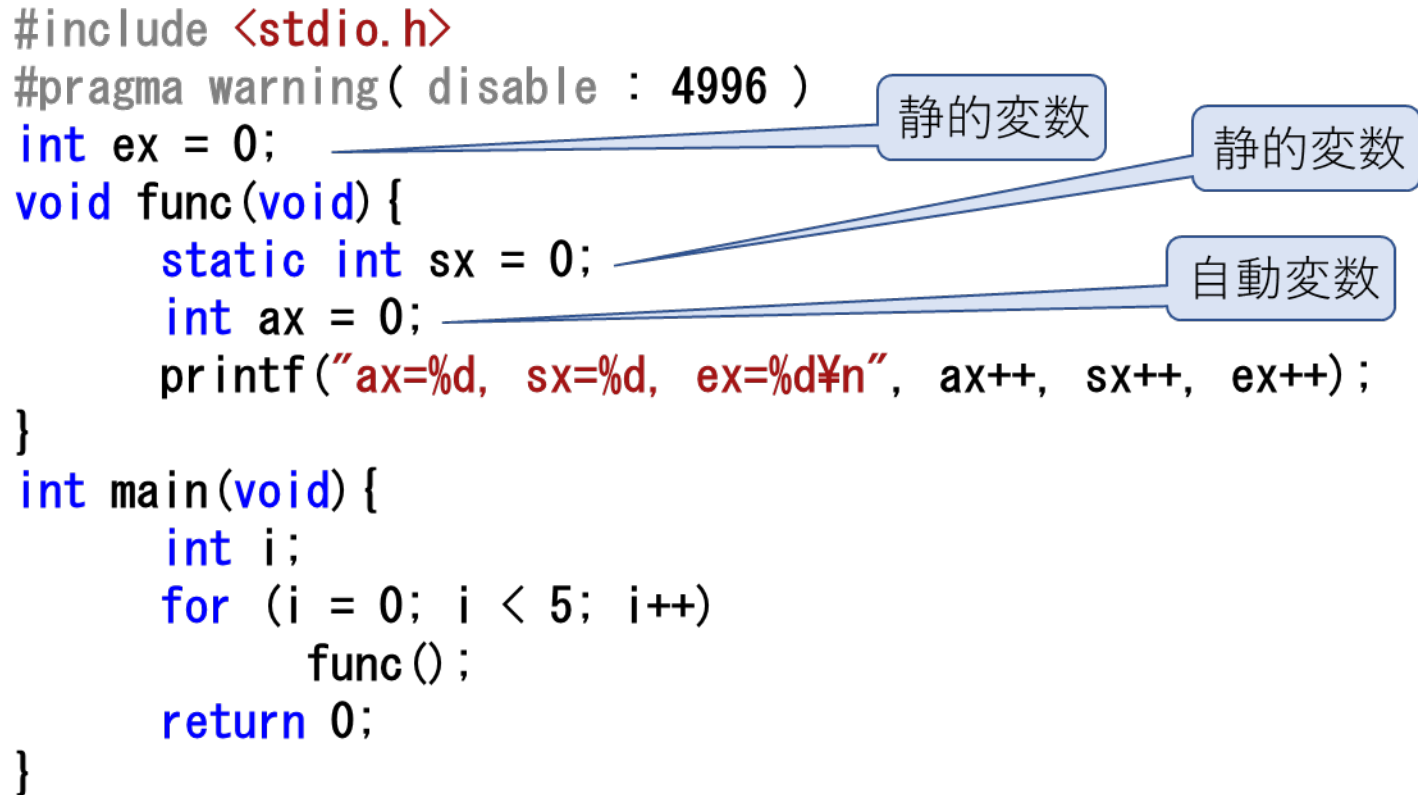
### ※ポイント：

- ・グローバル変数は0で初期化される
- ・グローバル変数はどこからでも呼び出せて便利
- ・各関数内でのローカル変数はそのスコープ内でのみ呼び出せる

# 記憶クラス(静的変数と自動変数)

## List2.3

```
#include <stdio.h>
#pragma warning( disable : 4996 )
int ex = 0;
void func(void) {
    static int sx = 0;
    int ax = 0;
    printf("ax=%d, sx=%d, ex=%d\n", ax++, sx++, ex++);
}
int main(void) {
    int i;
    for (i = 0; i < 5; i++)
        func();
    return 0;
}
```



```
ax=0, sx=0, ex=0
ax=0, sx=1, ex=1
ax=0, sx=2, ex=2
ax=0, sx=3, ex=3
ax=0, sx=4, ex=4
```

静的変数と自動変数の区別

- ・ **静的変数(List2.3のex, sx)**

スコープ：プログラム開始から終了まで  
初期化：0で初期化される

- ・ **自動変数(List2.3のax)**

スコープ：ブロック内  
初期化：されない(不定値)

※ポイント：ローカル変数の頭にstaticをつけて宣言をすると、グローバル変数と同じように振る舞う（つまり、静的変数の初期化はコンパイル時に1回のみ行われる）。

【例題】 静的変数を用いたカウンター

List2.4

```
#include <stdio.h>
#pragma warning( disable : 4996 )
void func(void) {
    static int count = 0;
    printf("[func] %d¥n", ++count);
}
int main(void) {
    int i;
    for (i = 0; i < 5; i++)
        func();
    return 0;
}
```

実行結果

```
[func] 1
[func] 2
[func] 3
[func] 4
[func] 5
```

## 演習問題2-1

A. List2.1において、ローカル変数Cの宣言`int x = i;`を代入文`x = i;`に変更した場合、実行結果がどのようなになるか確認せよ。また、その原因について考察せよ。

B. List2.3において、変数`ax`,`sx`,`ex`の初期化をそれぞれ省略した場合、どのような結果となるか確認せよ。また、その原因について考察せよ。

## 演習問題2-2

List 2.4を参考にして次のプログラムを作成せよ.

※外部変数とはグローバル変数のこと.

A. List 2-3の静的変数 `count` を外部変数に変更せよ.

B. `func()`と同様の関数`funcA()`と`funcB()`を作成し、`funcA()`の中で、`funcB()`を2回呼び出すように変更する. `funcA()`と`funcB()`が呼び出される回数をそれぞれ表示し、実行結果が図のようになるようにせよ.  
`countA`および`countB`は静的変数を用いよ.

```
[funcB] 1
[funcB] 2
[funcA] 1
[funcB] 3
[funcB] 4
[funcA] 2
[funcB] 5
[funcB] 6
[funcA] 3
[funcB] 7
[funcB] 8
[funcA] 4
[funcB] 9
[funcB] 10
[funcA] 5
```



# 演習問題2-3

外部変数の配列

```
int a[] = { 1,2,3,4,5 };
```

```
int b[] = { 6,7,8,9,10 };
```

がある時、この2つの配列の積和

$$c = \sum_{i=0}^4 a[i] \times b[i]$$

を計算する関数を設計し、結果を表示せよ。答えは130となる。