プログラミング第一同演習

慶應義塾大学 理工学部 情報工学科
講義担当: 河野 健二
演習担当: 杉浦 裕太

本日の内容

sslab system software

- 関数
  - 引数(仮引数，実引数)，戻り値
  - return 文
- 変数のスコープ (scope)
  - ローカル変数とグローバル変数
  - Python とは違う
- 値呼び出し (call-by-value)
- ポインタの基本（Python にはない概念）
  - ポインタ型
  - & と * の意味

char (文字) 型について

sslab system software

- ひと文字を覚えておくための変数
  - 0 文字以上の任意の長さの文字列とは違う
    - その辺のきちんとした話はいずれします
  - コンピュータの内部では文字も数値で表現されている
    - 文字 a は 97, 文字 b は 98,
      文字 A は 65, 文字 B 66 といった具合
  - char 型の変数には文字に対応する値が入っている
    - 実際, char 型の変数には -128〜127 までの整数が格納可
  - char 型: 8 bit (== 1 byte) の整数を格納できる
  - int 型: 32 bit (== 4 byte) の変数を格納できる

ASCIIコード表

sslab system software

	0x00	0x10	0x20	0x30	0x40	0x50	0x60	0x70
0x00	NULL	DLE	SP	0	@	P	`	p
0x01	SOH	DC1	!	1	A	Q	a	q
0x02	STX	DC2	“	2	B	R	b	r
0x03	ETX	DC3	#	3	C	S	c	s
0x04	EOT	DC4	$	4	D	T	d	t
0x05	ENQ	NAK	%	5	E	U	e	u
0x06	ACK	SYN	&	6	F	V	f	v
0x07	BEL	ETB	‘	7	G	W	g	w
0x08	BS	CAN	(	8	H	X	h	x
0x09	HT	EM	)	9	I	Y	i	y
0x0a	NL	SUB	*	:	J	Z	j	z
0x0b	VT	ESC	+	;	K	[	k	{
0x0c	NP	FS	,	<	L	¥	l	|
0x0d	CR	GS	-	=	M	]	m	}
0x0e	SO	RS	.	>	N	^	n	~
0x0f	SI	US	/	?	O	_	o	DEL

赤字: 制御文字
青地: スペース
黒字: 通常文字

char (文字) 型について

sslab system software

- 使い方
  - `char c;` // ひと文字を覚えておくための変数 c を宣言

```
c = 'a';   // 変数 c に文字 a を代入する．' で囲む！

printf("c = %c¥n", c);   // 変数 c の値を表示する
scanf("%c", &c);         // 変数 c に値を入力する
```

関数

- 数学で使う関数とちょっと似ている
  - 数学の例：
    $f(x) = x^2 + x + 1$ と定義する
    $f(1), f(2), f(3)$ とすると，
    それぞれ，$1^2 + 1 + 1, 2^2 + 2 + 1, 3^2 + 3 + 1$ が計算され，3, 7, 13 が返る
- C 言語でも関数が定義できる
  - 次のような感じ（詳細は次のスライド）

```
int f(int x)
{
    return x * x + x + 1;
}
```

関数の定義

- 引数の型を指定する
  - 右の例では int 型の引数をとることを指定
- 戻り値の型を指定する
  - 右の例では int 型の戻り値を返すことを指定
- return 文を使って明示的に値を返す
- main() の外で定義する

青字の int は戻り値の型が int であることを示す

赤字の int は引数 x の型が int であることを示す

```
int f(int x)
{
    return x * x + x + 1;
}

int main()
{
    …
}
```

return で値を返す
戻り値の型と一致させること

関数の呼び出し方

- 引数を指定して呼び出す
  - 引数の型が一致している必要がある
- 戻り値として指定した型の値が戻ってくる

```
int f(int x)
{
    return x * x + x + 1;
}

int main()
{
    int y = f(1);
    printf("f(1) = %d¥n", y);
    printf("f(2) = %d¥n", f(2));

    return 0;
}
```

f(1) の値，すなわち 3 が返ってくる
戻り値の型は int なので，int 型の
変数に戻り値を代入

f(2) の値，すなわち 7 が返ってくる
戻り値の型は int なので，%d を
使って結果を表示

関数の例（その1）

- 絶対値を返す関数

今まで習ったプログラムが書ける

```
int absolute(int x)
{
    if (x > 0) {
        return x;
    } else {
        return -x;
    }
}

int main()
{
    int y = absolute(1);
    printf("absolute(1) = %d¥n", y);
    printf("absolute(-1) = %d¥n", absolute(-1));
    return 0;
}
```

return 文は 2 箇所以上あってもよい
- return 文を実行すると，そこで関数の実行は終了する（詳細は後で）

関数の例（その2）

- 階乗を返す関数

```
int factorial(int n)
{
    int i, r = 1;
    for (i = 2; i <= n; i++) {
        r *= i;
    }
    return r;
}

int main()
{
    int y = factorial(3);
    printf("factorial(3) = %d¥n", y);
    printf("factorial(0) = %d¥n", factorial(0));
    return 0;
}
```

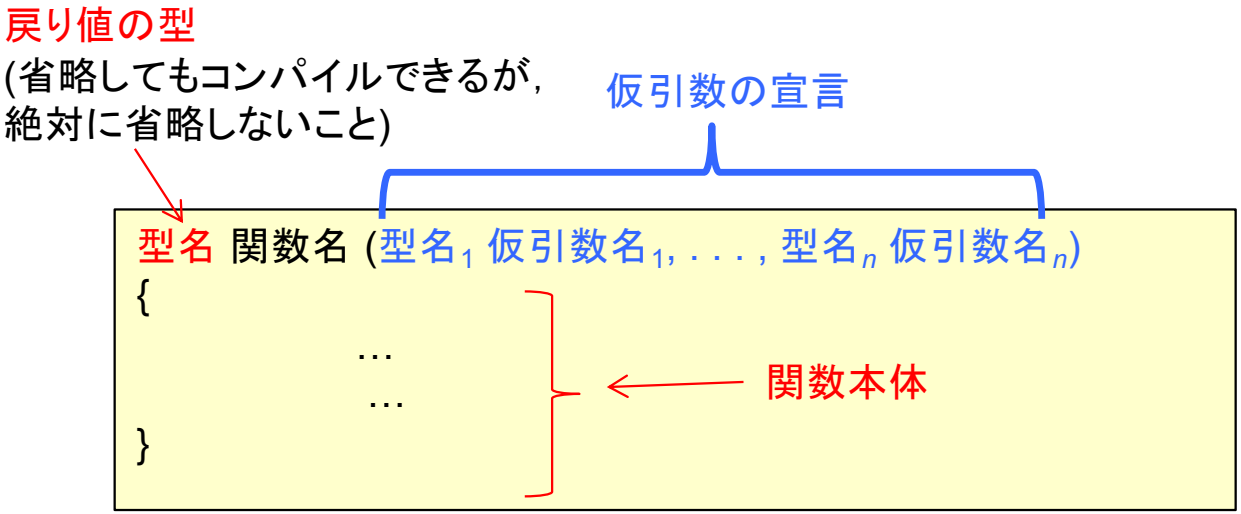
変数も宣言できる
（詳細は後で）

実は main() も関数のひとつ
int 型を返す

ここで int 型の値 0 を
返している

関数定義の詳細

- 仮引数 (parameter): 関数定義に用いられる引数
  - “$f(x) = x^2 + 5x + 6$” の ‘x’ にあたるもの
- 実引数 (argument): 実行時に関数に渡される引数
  - $f(3)$ として $f(x)$ を呼び出す時の 3 にあたるもの



return 文の詳細

- 一般形: return 式;
  - ただし，値を返さない関数（詳細は後述）のときは式は書かない

- 関数の戻り値を指定すると同時に，関数の終了を表す
  - return 文を実行すると，関数内のその後の処理は実行されない

```
int foo(int x)
{
    return x + 1;
    printf("not reached¥n");
}
```

return x + 1 を実行すると，関数 foo() の実行は終了する．したがって，この printf() は実行されない

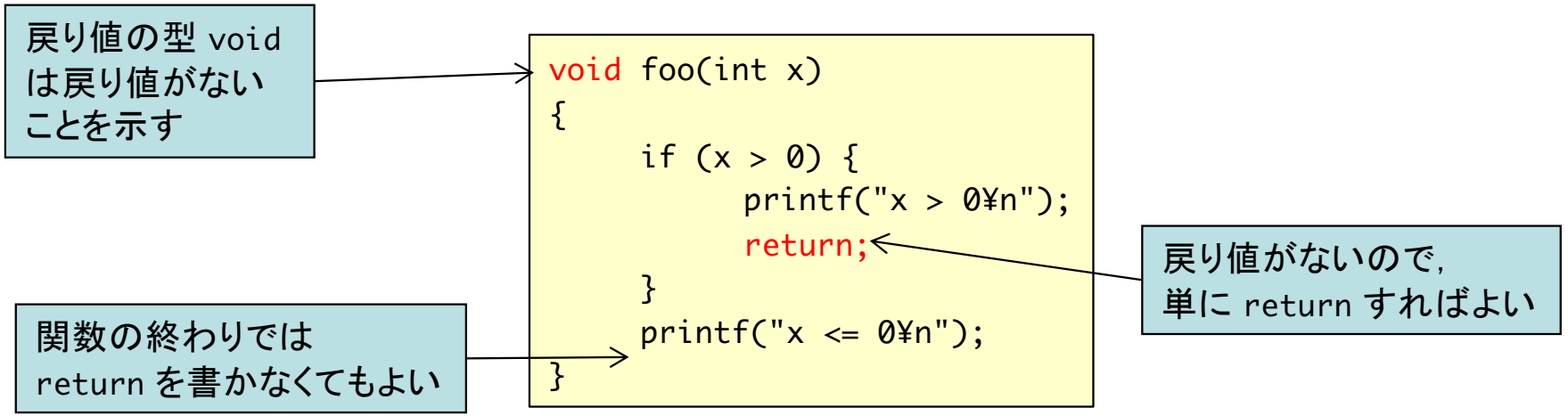
おまけ

sslab system software

- `main()` 関数の `return 0;` はどこに値を返すのか?
  - シェル (shell) に値を返す (シェルについては他でやってますよね?)
  - 0 は正常終了したという意味
  - 異常終了の場合は別の値を返すようにする (本科目では扱わない)

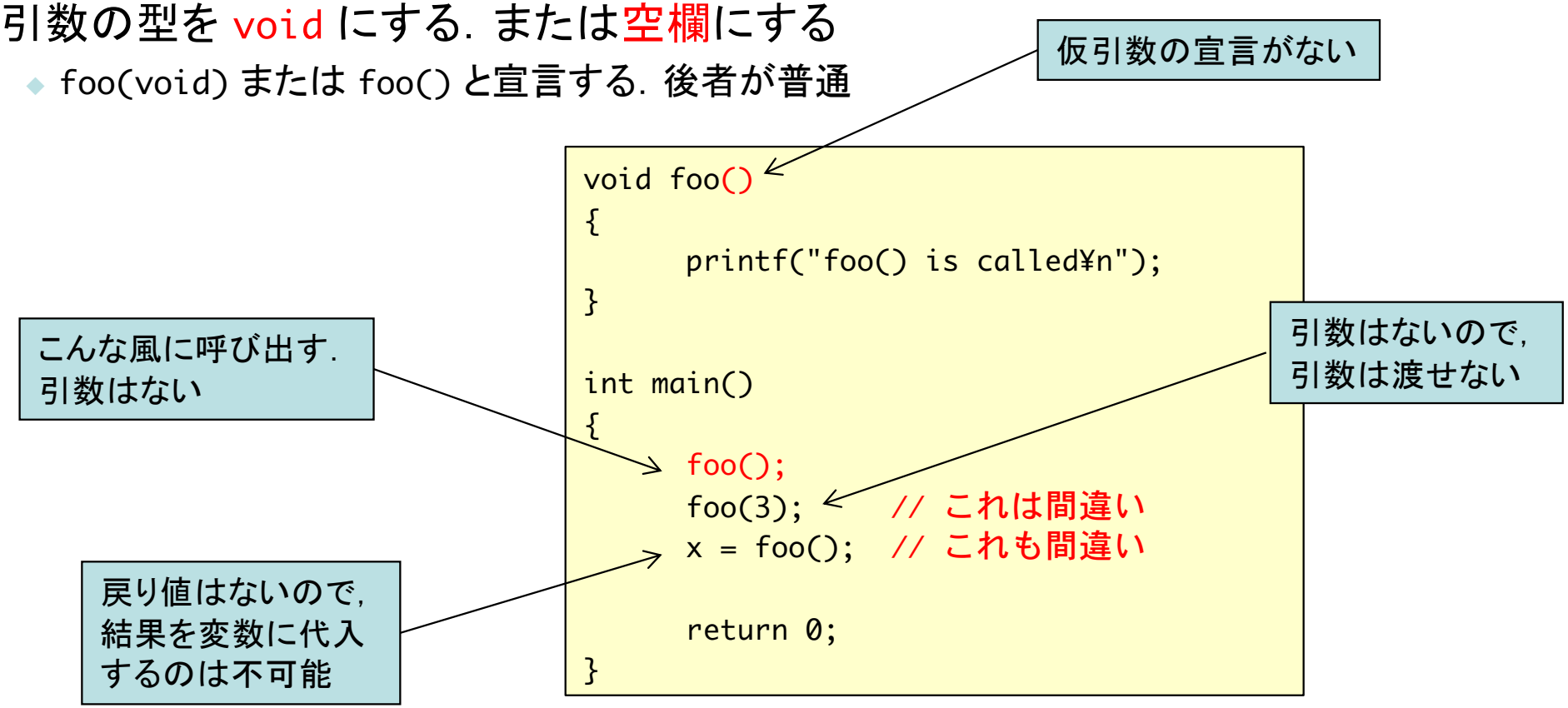
void 型（その1）

- 戻り値を返さない関数も定義できる
  - 戻り値の型を void にする
    - void は戻り値がないことを示す
  - 戻り値がないので，単に return; で関数の実行を終わる
    - 関数の終わりに到達するときは return; も書かないでよい



void 型(その2)

- 引数をとらない関数も定義できる
  - 引数の型を void にする．または空欄にする
    - foo(void) または foo() と宣言する．後者が普通



ローカル変数とスコープ (scope)

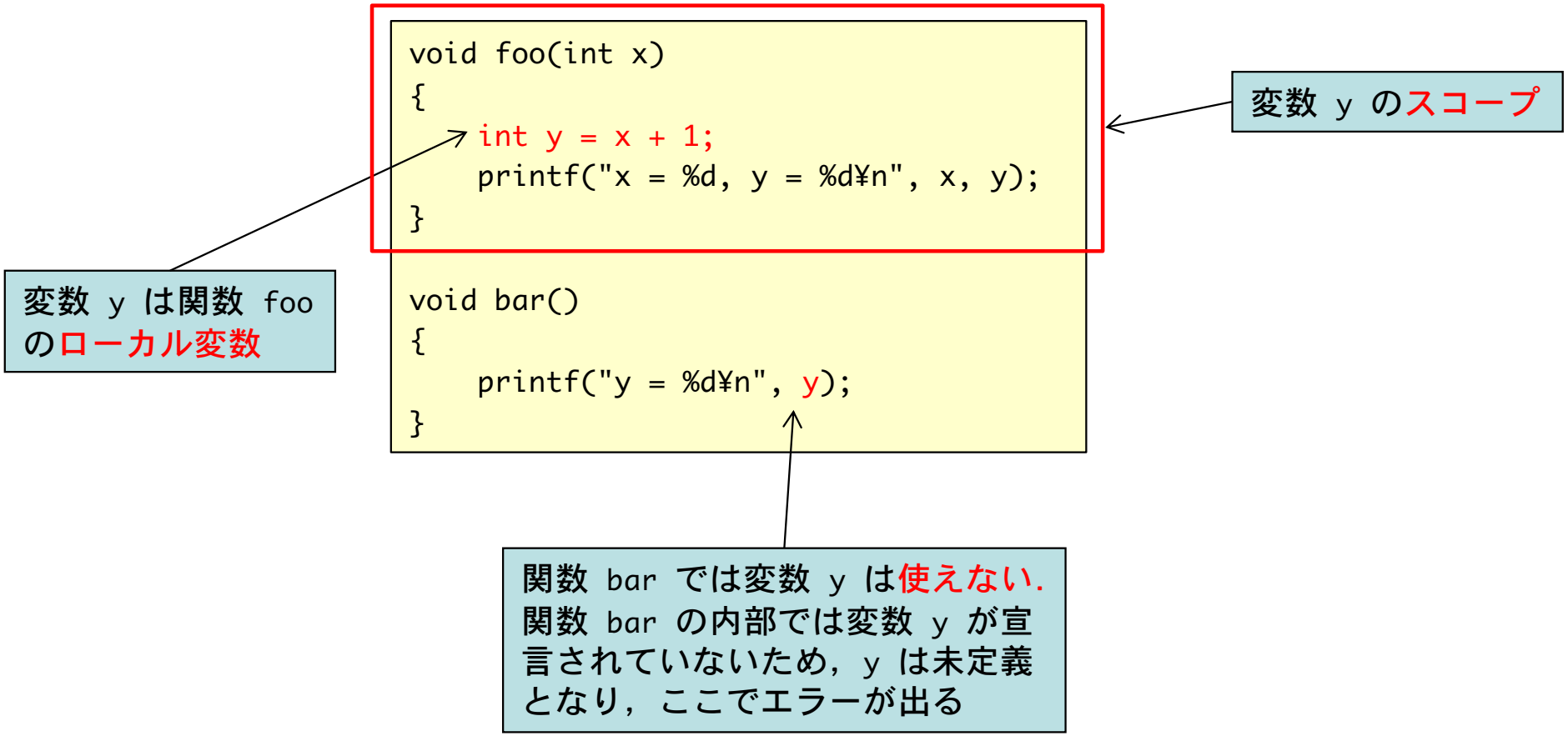
- 関数内で定義された変数のことを
  ローカル変数 (local variable) または局所変数という
  - 他にも大域変数 (global variable) というものがある
- ローカル変数は定義された関数内でのみ利用可能
  - 変数が利用できる範囲のことをスコープ (scope) という

```
int factorial(int n)
{
    int i, r = 1;
    for (i = 2; i <= n; i++) {
        r *= i;
    }
    return r;
}
```

i と r は関数 factorial の中で定義されている．
これらの変数のことをローカル変数または局所変数という

変数 i と r のスコープ．
i と r はこの関数内でのみ使える

ローカル変数のスコープ (1/2)



ローカル変数のスコープ (2/2)

関数が違えば，同じ名前の変数も宣言できる．
この変数 y は foo の中の y とは別物

```
void foo(int x) {
    int y = x + 1;
    printf("x = %d, y = %d¥n", x, y);
}


void bar() {
    int y = 100;
    printf("y = %d¥n", y);
}

int main() {
    foo(3);
    bar();
    …
```

この y の値を変更しても，
foo の中の y の値は変わらない

グローバル変数とスコープ (scope)

- 関数外で定義された変数のことを
  グローバル変数 (global variable) または大域変数という
- グローバル変数はすべての関数から利用可能
  - 厳密には違う．“分割コンパイル”の話をするときに説明

変数 z のスコープ

foo, bar の中で z を使うことができる

```
foo(3);
bar();
```
と呼び出すと，foo の中で
x = 3, z = 4
と表示され，bar の中で
z = 4
と表示される

```
int z;    // グローバル変数の宣言

void foo(int x) {
    z = x + 1;
    printf("x = %d, z = %d¥n", x, z);
}

void bar() {
    printf("z = %d¥n", z);
}
```

グローバル変数とローカル変数 (注意点)

- 変数名の衝突
  - グローバル変数と同じ名前のローカル変数は避ける

```
int z = 0;  // グローバル変数．0 に初期化

void foo(int x) {
    int z = x + 1;
    printf("x = %d, z = %d¥n", x, z);
}

void bar() {
    printf("y = %d¥n", z);
}

int main() {
    foo(3);
    bar();
    return 0;
}
```

グローバル変数 z を宣言

ローカル変数 z を宣言
グローバル変数 z を隠してしまう

これはローカル変数 z を参照

これはグローバル変数 z を参照

foo の中で
x = 3, z = 4
と表示され，bar の中で
z = 0
と表示される

グローバル変数とローカル変数(注意点)

- 変数は初期化してから使う
  - ローカル変数ではどんな値が入っているのか不明
    - 多くの場合，0 が入っているが，そんな保証はない
  - グローバル変数は 0 で初期化されているが，初期化してから使う方がよい

このプログラムは誤り．
変数 sum の値が 0 に初期化されていない．
偶然，sum の初期値が 0 の時しか正しい結果にならない．

```
int i, sum = 0;
```

とすること

```
int sum(int n) // 1 から n の合計を求める
{
    int i, sum;
    for (i = 1; i <= n; i++) {
        sum += i;
    }
    return sum;
}
```

Python との違い (注意点)

sslab system software

- 関数の中で関数を定義することはできない
  - すなわち，入れ子の関数は定義できない
- スコープルールも少し違う
  - Python の nonlocal, global に相当するものはない
- 配列を引数に渡す方法は少し難しい
  - ポインタについての理解を必要とする
  - もう少し先の講義で教える

Quiz

```
int z = 1;

void f(int x)
{
    int z = x;
    printf("f: z = %d¥n", z);
}

int g(int x)
{
    z += x;
    printf("g: z = %d¥n", z);
}

int h(int z)
{
    z += 1;
    printf("h: z = %d¥n", z);
}
```

```
int main()
{
    f(2);
    printf("main: z = %d¥n", z);

    g(3);
    printf("main: z = %d¥n", z);

    h(5);
    printf("main: z = %d¥n", z);

    return 0;
}
```

- 実行結果はどうなる？
  - »
```
f: z = 2
main: z = 1
g: z = 4
main: z = 4
h: z = 6
main: z = 4
```

仮引数と実引数の関係

- 仮引数には実引数のコピーが渡される
  - これを値渡し (call-by-value) という
    - C 言語では call-by-value しかない
    - C++ という言語には参照渡し (call-by-reference) というのもある

```
void foo(int x)
{
    x = x + 3;
}

int main()
{
   int y = 4;
   foo(y);
   printf("y = %d¥n", y);
   return 0;
}
```

(仮)引数の値を更新する
仮引数には実引数のコピーが入っている

よって，実引数 y の値は 4 のまま

参考：C++ における参照渡し

```
void foo(int& x) {
   x = x + 3;
}
int main() {
   int y = 4;
   foo(y);
   printf("y = %d¥n", y);
   return 0;
}
```

参照渡し．
Cでは使えない

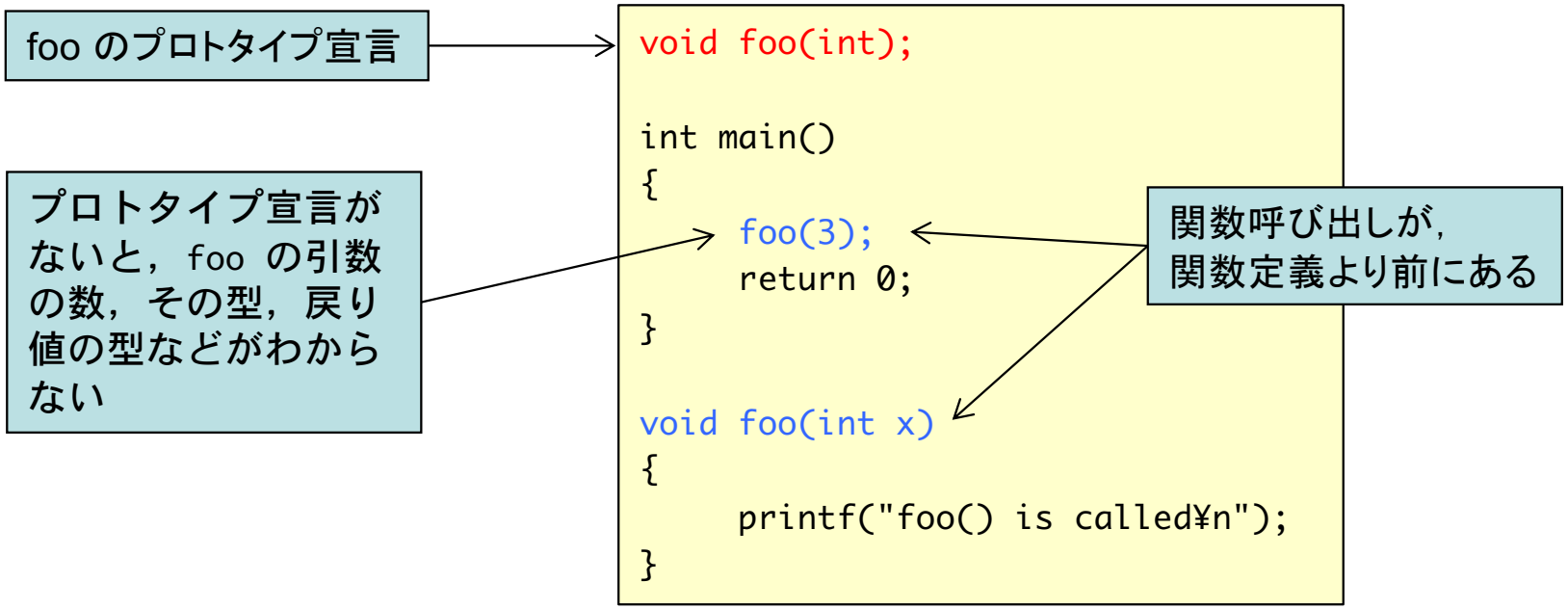
y の値は 7 になる

関数のプロトタイプ宣言

- あらかじめ，ある関数を使うことを宣言しておくもの
  - 関数定義より前に関数の呼び出しがある場合など

型名 関数名(型名$_1$, . . . , 型名$_n$);

- 仮引数の型を並べる（仮引数名は省略してもよい）
- 仮引数が無い場合は空白、または void と書く



Quiz

- 実行結果はどうなるでしょう？
  - w = 101, x = 401, y = 201
    w = 102, x = 401, y = 201
    x = 400

```
int w = 100;
void f(int x)
{
    int y = 200;
    w++; x++; y++;
    printf("w = %d, x = %d, y = %d¥n", w, x, y);
}
int main()
{
    int x = 400;
    f(x);
    f(x);
    printf("x = %d¥n", x);
    return 0;
}
```

例題: 棒グラフ

- 標準入力から順次得点(整数値)を読み込んで，その得点分だけ星印(*)を横方向に並べた棒グラフを作りなさい．

  ただし，負の値を入力した時点で得点の入力を終わるとする．

```
1
   1:*
0
   0:
49
  49:**************************************************
50
  50:**************************************************
51
  51:**************************************************
     *
102
 102:**************************************************
     **************************************************
     **
15
  15:***************
-1
```

例題: プログラム

```c
#include <stdio.h>

#define WIDTH 50

void graph(int);

int main()
{
    int score;

    scanf("%d", &score);
    for(; score >= 0;) {
        graph(score);
        scanf("%d", &score);
    }

    return 0;
}
```

マクロの定義

プロトタイプ宣言

```c
void
graph(int len)
{
    int i;

    printf("%4d:", len);
    for (i = 0; i < len; i++) {
        if (i > 0 && i % WIDTH == 0)
            printf("¥n    ");
        printf("*");
    }
    printf("¥n");
    return;
}
```

値を返さない関数

マクロの利用
50 に置き換えられる

この return はなくてもよい

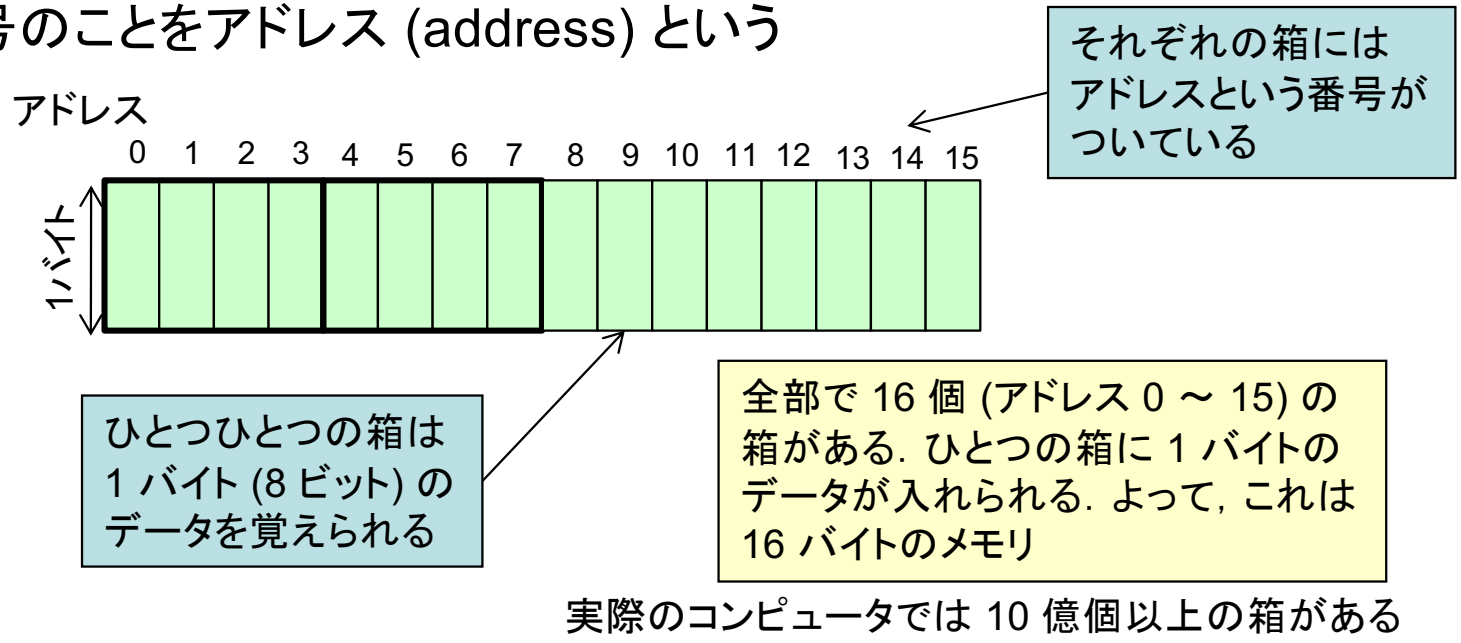
ポインタの基礎

sslab system software

- ポインタとは？
  - すでに存在するデータを指し示すためのデータ
  - 難しいとよく言われるが，原理を理解すれば簡単
- まずは変数とメモリの関係を理解しよう
  - 変数とは何だったか？
    - 変数は値を「覚えて」おくためのもの
  - コンピュータは「メモリ」を使ってデータを覚える
    - したがって，変数も「メモリ」に格納されている

メモリの仕組み

- メモリとは，データを覚えておくためのもの
  - 8 bit の 2 進数，すなわち 1 バイト単位でデータを記憶する
  - 1 バイトのデータを覚えておくための箱がずらりと並んでいる
  - それぞれの箱には 0 から順に番号がついている
  - この番号のことをアドレス (address) という

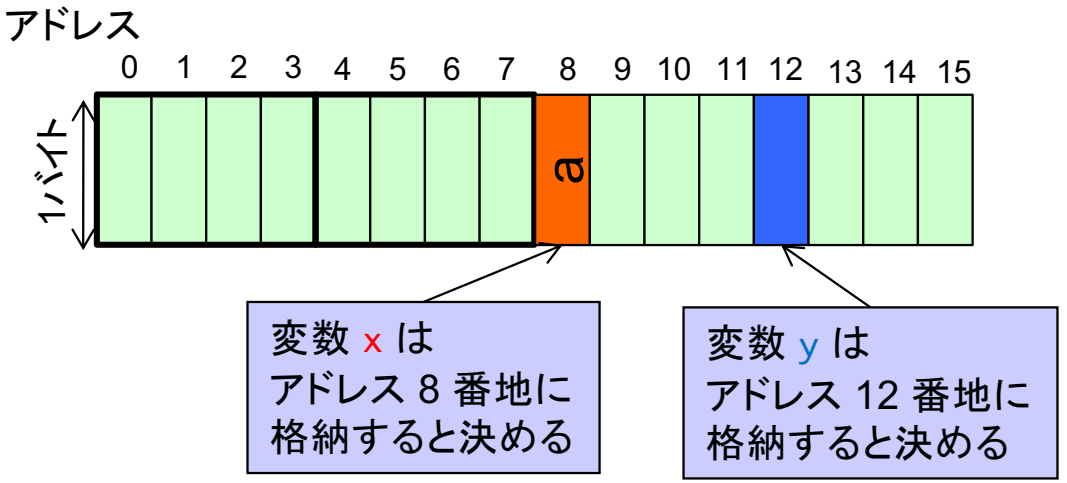


変数とメモリの関係：その 1

- 変数をメモリに入れるためには？
  - C 言語の処理系が勝手に次のことをやってくれている
    - それぞれの変数について，値を入れておくためのアドレスを決める
    - 変数への読み書きがあると，アドレスを指定して読み書きする
- すべての変数にはアドレスが対応づけられている
  - すべての変数はメモリ上のどこかに割り当てられている

```
int main()
{
    char x;
    char y;

    x = 'a';
    printf("%c", x);
    …
}
```

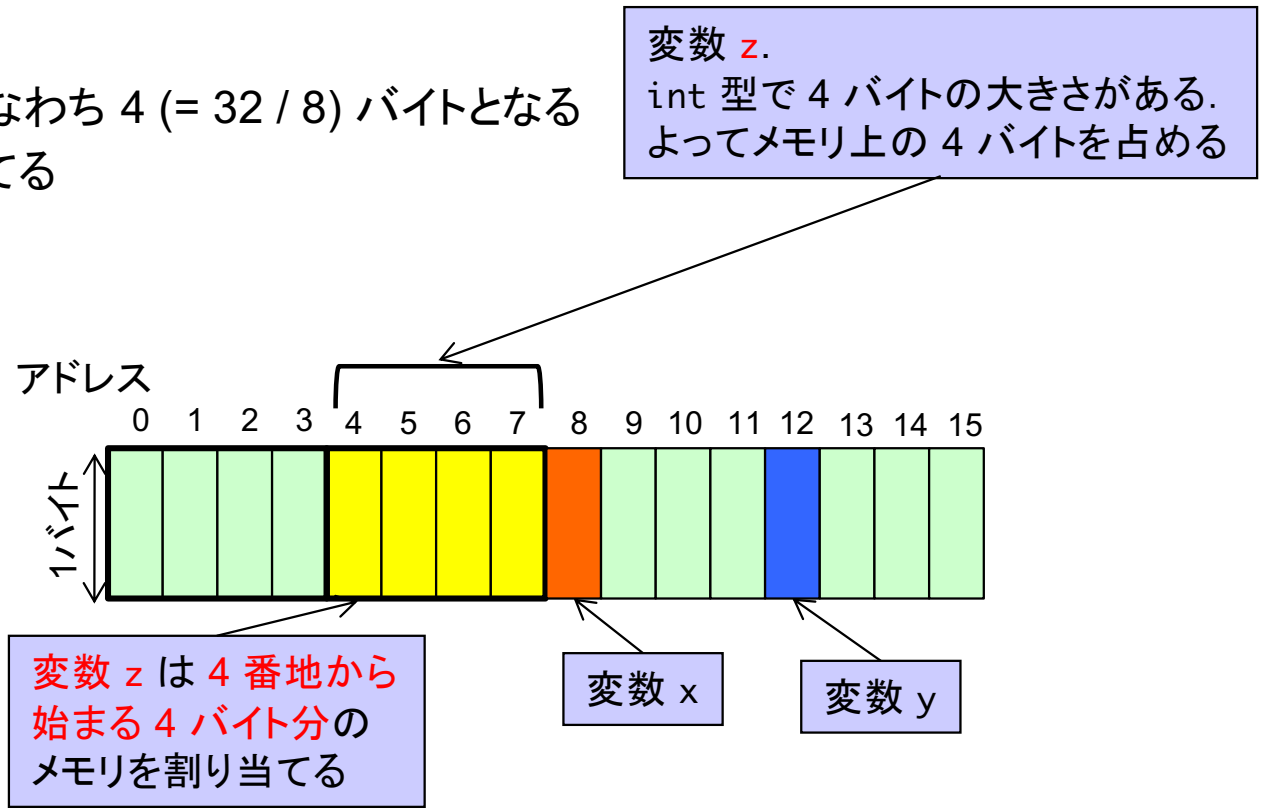
変数 x への読み書きはアドレス8番地への読み書きとなる



変数とメモリの関係: その 2

- もう少し複雑な場合を考える
  - char 型の変数は 1 バイト
  - int 型の変数は 4 バイト
    - int 型は 32 bit の整数. すなわち 4 (= 32 / 8) バイトとなる
    - 4 バイト分のメモリを割り当てる

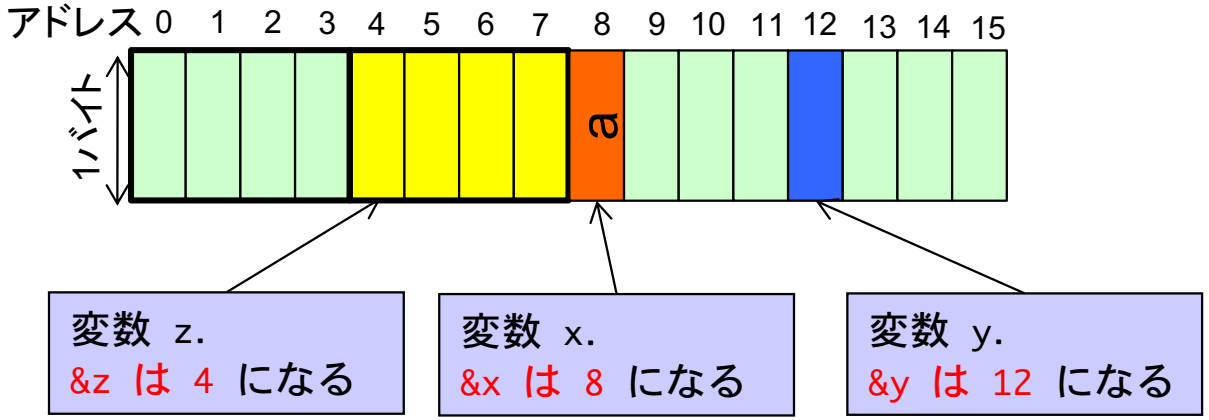
```
int main()
{
    char x;
    char y;
    int z;
    …
}
```



ポインタとは?

- ポインタとは変数が格納されているアドレスのことである
  - 少し不正確だが，この理解で困ることは絶対にない
- C 言語では，変数が格納されているアドレスを取得できる
  - &x と書くと，変数 x が格納されているアドレスが得られる
  - &z と書くと，変数 z が格納されているアドレスが得られる
    - 変数 z はアドレス 4〜7 の 4 バイトを占めるが，&z はその先頭アドレスになる

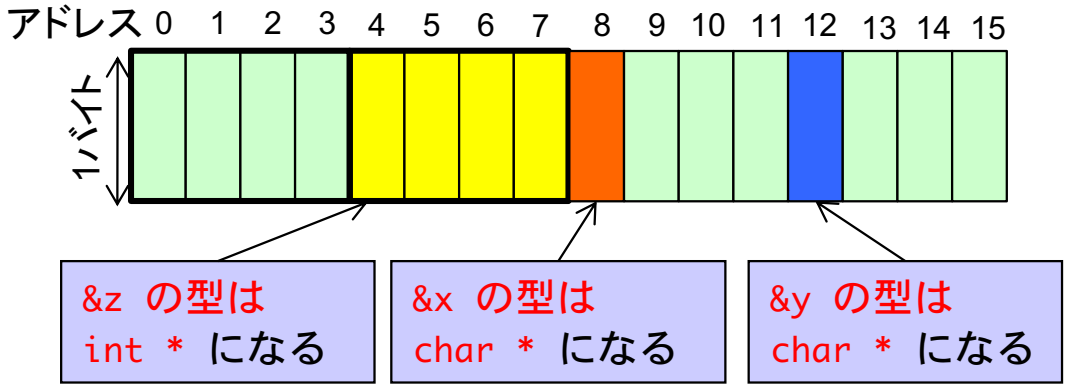
```
int main()
{
    char x;
    char y;
    int z;
    …
}
```



ポインタ型

- さきほどの &x, &z の型は何になるのだろうか？
  - &x は char 型の変数を指すアドレス
    - アドレス 8 番地に置かれているのは char 型の変数
    - そこを指すアドレスは “char *” という型になる
  - &z は int 型の変数を指すアドレス
    - アドレス 4 番地に置かれているのは int 型の変数
    - そこを指すアドレスは “int *” という型になる

```
int main()
{
    char x;
    char y;
    int z;
    …
}
```



ポインタの使い方

sslab system software

- 変数のアドレスをポインタ型の変数に代入する
  - `int x = 40;`
    `int *p;  // 変数 p には int 型の変数が置かれているアドレスが入る`
    `p = &x;  // 変数 p には変数 x のアドレスが入る`

- ポインタの参照するアドレスの値を読み書きする
  - 変数の前に * をつける
    - `printf("%d", *p);`
      ポインタ p の指している先の値を表示
    - `*p = 100;`
      ポインタ p の指している先に 100 を代入

ポインタを使ったプログラム例

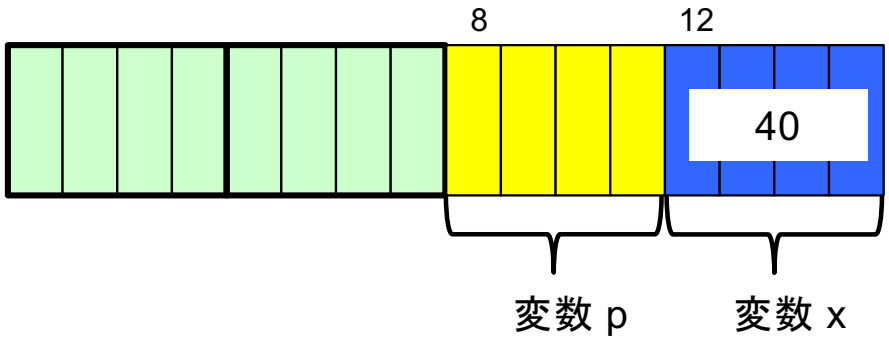
- 次のプログラムを実行すると・・・

```
int main()
{
    int x = 40, *p;

    p = &x;
    printf("%d", *p);

    *p = 100;
    printf("%d¥n", x);

    return 0;
}
```

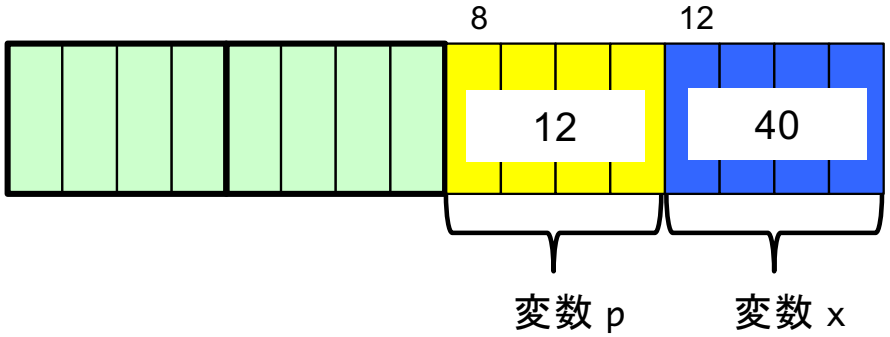
  - 40
    100
    と表示される．その理由を考えてみよう

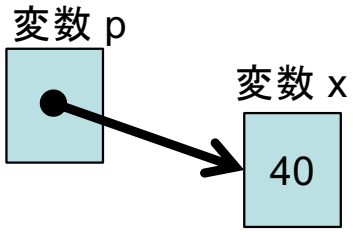
ポインタを使ったプログラム例

- `int x = 40, *p;`
  - メモリのどこかに変数 x と 変数 p が割り当てられる
    - 下の例ではアドレス 12 番地に x, アドレス 8 番地に p.
  - 変数 x の値は 40 に初期化される



  - `p = &x;`
    - 変数 p に変数 x のアドレスが入る

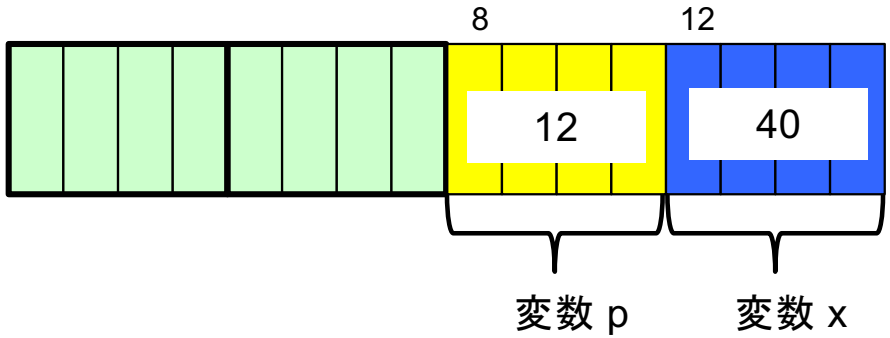


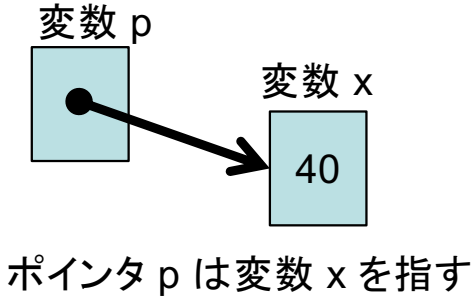


ポインタ p は変数 x を指す

ポインタを使ったプログラム例

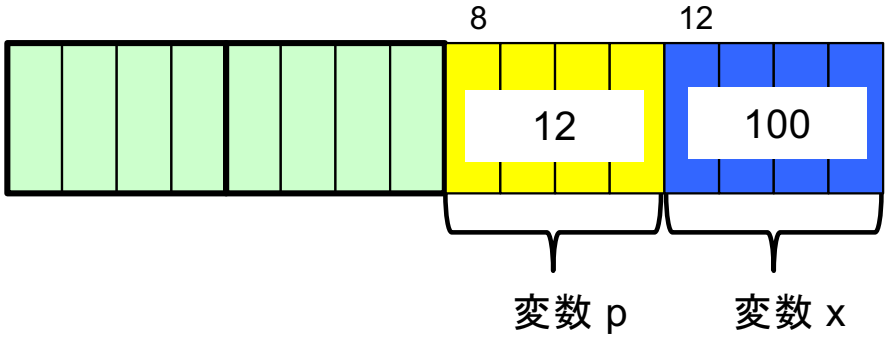
- `printf("%d¥n", *p);`
  - 変数 p に入っているアドレスのところに置かれている値を表示する
  - ポインタ変数 p の値は 12
  - よって，アドレス 12 番地に格納されている値 40 を表示する
  - アドレス 12 番地に格納されているのは変数 x.
    よって，変数 x の値を表示する

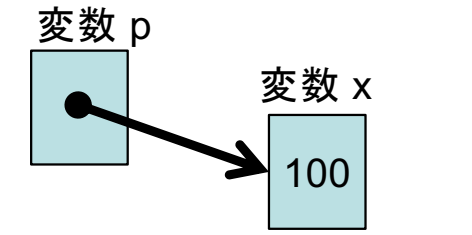




ポインタを使ったプログラム例

- `*p = 100;`
  - 変数 p に入っているアドレスのところに 100 を代入する
  - ポインタ変数 p の値は 12
  - よって，アドレス 12 番地に 100 を代入する
  - アドレス 12 番地に格納されているのは変数 x.
    よって，変数 x に 100 を代入する





ポインタ p は変数 x を指す

例題: 変数の値の入れ替え

- ふたつの変数を入れ替える関数を作ろう
  - 変数の値を入れ替えることをスワップ (swap) という
- 次の関数でうまくいく？ うまくいかない・・・

```
void swap(int x, int y)
{
    int t;
    t = x;
    x = y;
    y = t;
}
int main()
{
    int a = 10, b = 20;
    swap(a, b);
    printf("a = %d, b = %d¥n", a, b);
    return 0;
}
```

変数 t を使わずに，
x = y;
y = x;
ではうまくいかないことに注意

a = 10, b = 20
と表示される．

C 言語は call-by-value.
関数の中で引数の値を
更新しても，それは呼び出し
元には反映されない

例題：変数の値の入れ替え

- 入れ替えたい変数へのポインタを渡すようにする

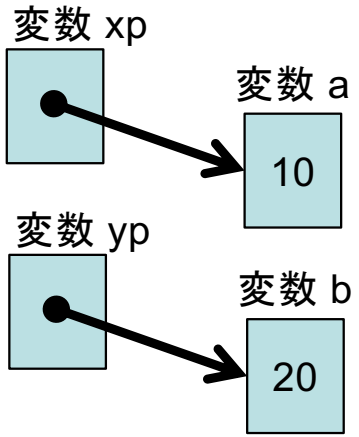
```
void swap(int *xp, int *yp)
{
    int t;
    t = *xp;
    *xp = *yp;
    *yp = t;
}

int main()
{
    int a = 10,  b = 20;
    swap(&a, &b);
    printf("a = %d, b = %d¥n", a, b);
    return 0;
}
```

変数 a のアドレス，
変数 b のアドレスを
受け取るようにする．

変数 a のアドレス，
変数 b のアドレスを
swap() に渡す

今度はうまくいく．
a = 20, b = 10
と表示される．

swap() が呼び出された時点では，
次のようになっている



要するに・・・

sslab system software

- C 言語には call-by-value しかない
- call-by-reference を実現するには？
- ポインタを使って call-by-reference を実現する
  - 正確には・・・
    call-by-reference のある言語では，
    処理系の内部でポインタ渡しで実現している

Quiz: 実行結果は？ (その1)

sslab system software

```
int main()
{
    int x = 40, *p;
    p = &x;
    x = 100;
    printf("%d", *p);
    return 0;
}
```

答え：100

```
int main()
{
    int x = 40, *p, *q;
    p = &x;
    q = p;
    *p = 200;
    printf("%d", *q);
    return 0;
}
```

答え：200

```
void foo(int *p)
{
    *p = 54;
}

int main()
{
    int x = 40;
    foo(&x);
    printf("%d", x);
    return 0;
}
```

答え：54

Quiz: 実行結果は？ (その2)

```
void bar(int *p)
{
    int *q = p;
    *q = 54;
}

int main()
{
    int x = 40;
    bar(&x);
    printf("%d", x);
    return 0;
}
```

答え：54

```
void f(int *p, int *q)
{
    *p = 10;
    *q = 20;
}

int main()
{
    int x = 40;
    f(&x, &x);
    printf("%d", x);
    return 0;
}
```

答え：20

まとめ

- 関数とスコープ・ルール
  - 微妙に Python と異なるので注意すること
  - 配列を引数にする方法は，もっと先で学ぶ
- ポインタ
  - ポインタはアドレスである
  - & 演算子と * 演算子
- ポインタはこの先，どんどん出てきます
  - 最初でつっかえないこと