

# プログラミング基礎

<http://bit.ly/prog2d>

## アドレスとポインタ

後期 第2週

2017/10/2

# メモリとアドレス

コンピュータはデータを格納するために、**記憶装置（メモリ）**を利用します。メモリは格納する場所を区別するために**アドレス（番地）**という番号が付けられています。

アドレス

0x1000 0x1001 0x1002 0x1003 0x1004 0x1005 0x1006 0x1007

メモリ

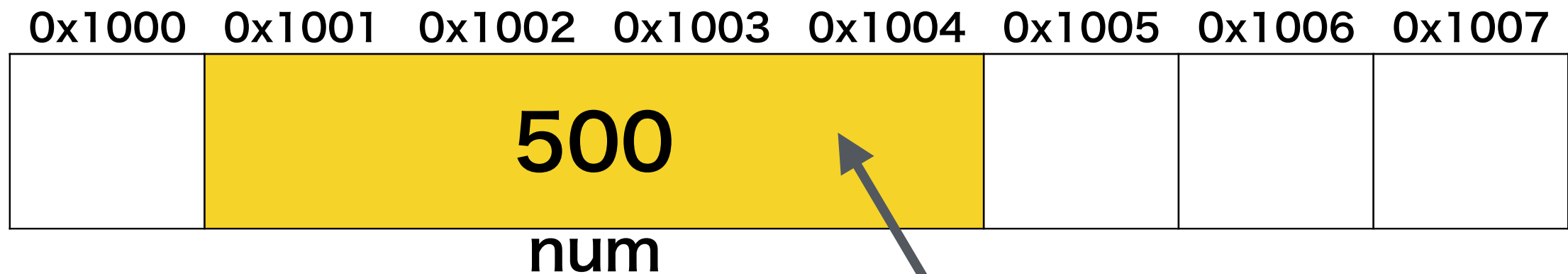
アドレスは順番通りにメモリに割り振られている  
(p.272)

# 変数とアドレス

変数を宣言すると、メモリ上にデータを格納する**場所**を**確保**します。プログラム中では、確保した場所を**変数名**で**指定**することができるようになります。

【例】

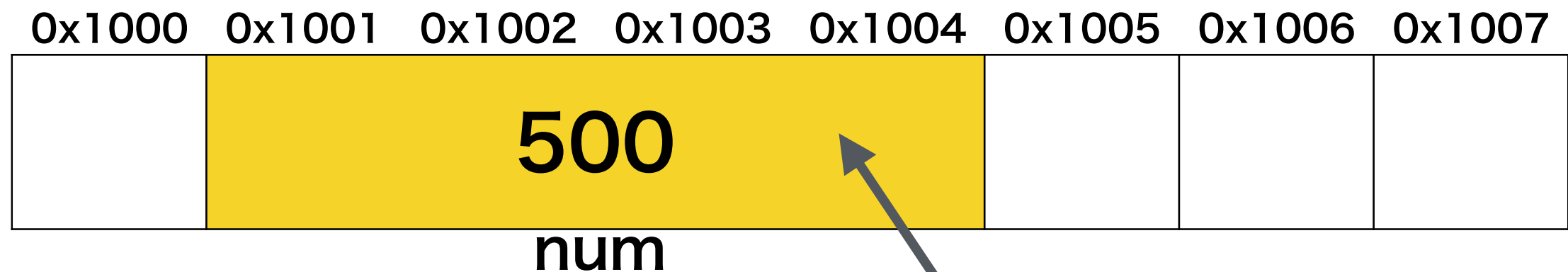
```
int num;  
num = 500;
```



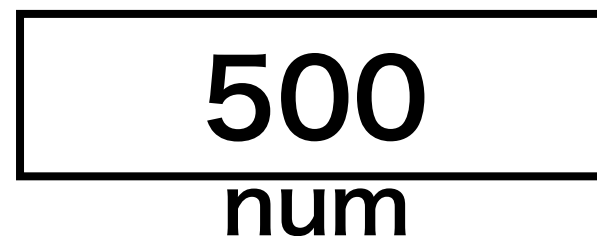
0x1001～1004というアドレスに格納場所（int型なので4バイト分）が確保された。メモリ上で空いている場所に確保されるため、実際には確保される場所は実行するたびに変化する。

# 変数のイメージ

プログラム内では「**num**という名前の格納場所に**500が入った**」ということがわかれば良いので、普段は、変数は「**箱**」のイメージで考えています。



実際には、メモリ上でこのようになっていても...



numという箱に入っているイメージで考えている

# アドレス演算子

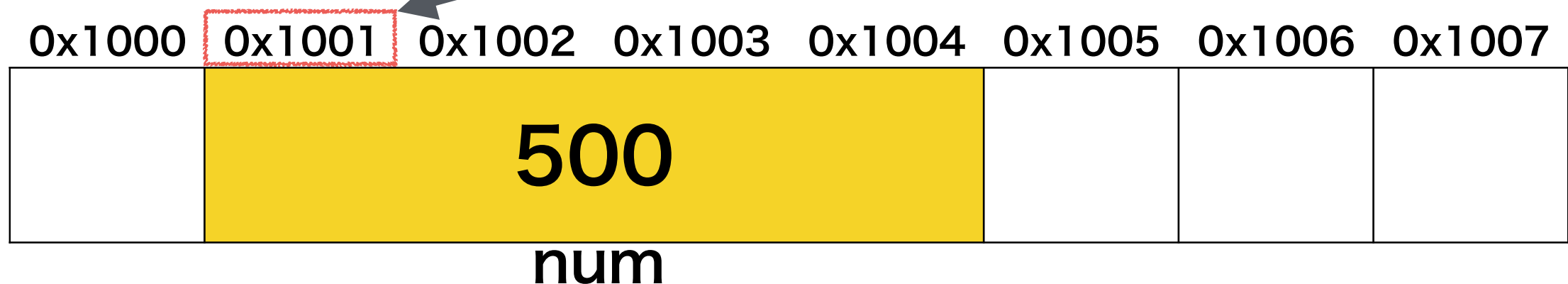
変数の格納場所のアドレスを、扱うには「**&変数名**」のように**アドレス演算子**「&」を使います。(p.273)

## 【例1】

```
printf("numのアドレス: %p\n", &num);
```

アドレスを出力する変換仕様には%pを指定する (p.274)

アドレス演算子でその変数の**先頭アドレス**が取得できる (p.275)

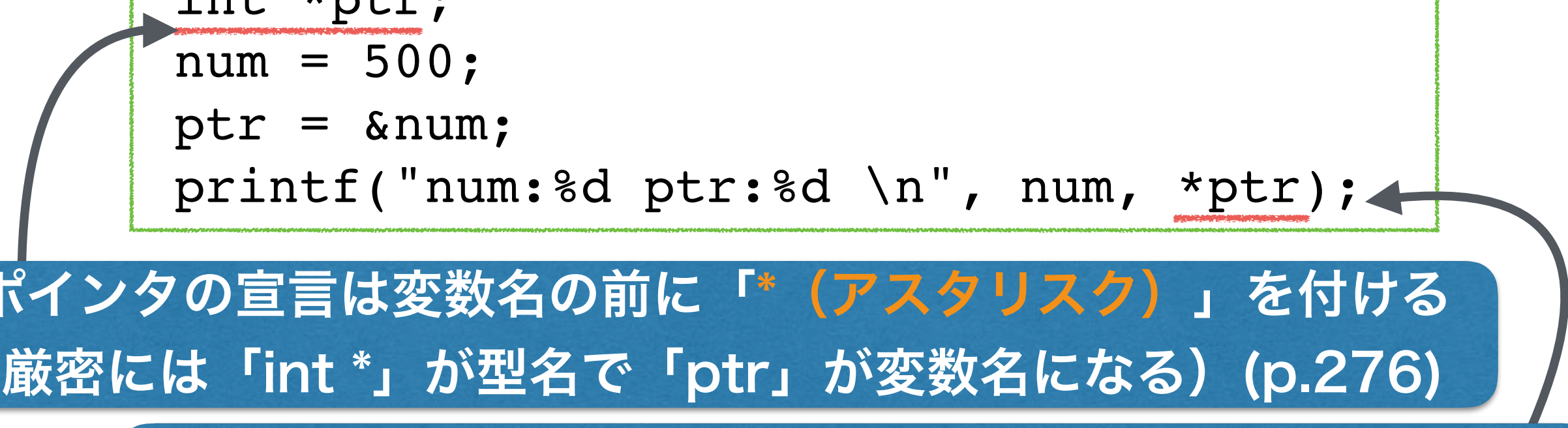


# ポインタ

「アドレスを格納する変数」をポインタ変数（ポインタ）と呼び、ポインタとしてメモリ上に確保された場所には、アドレスが格納されます。

## 【例2】

```
int num;  
int *ptr;  
num = 500;  
ptr = &num;  
printf("num:%d ptr:%d \n", num, *ptr);
```



ポインタの宣言は変数名の前に「\*（アスタリスク）」を付ける（厳密には「int \*」が型名で「ptr」が変数名になる）(p.276)

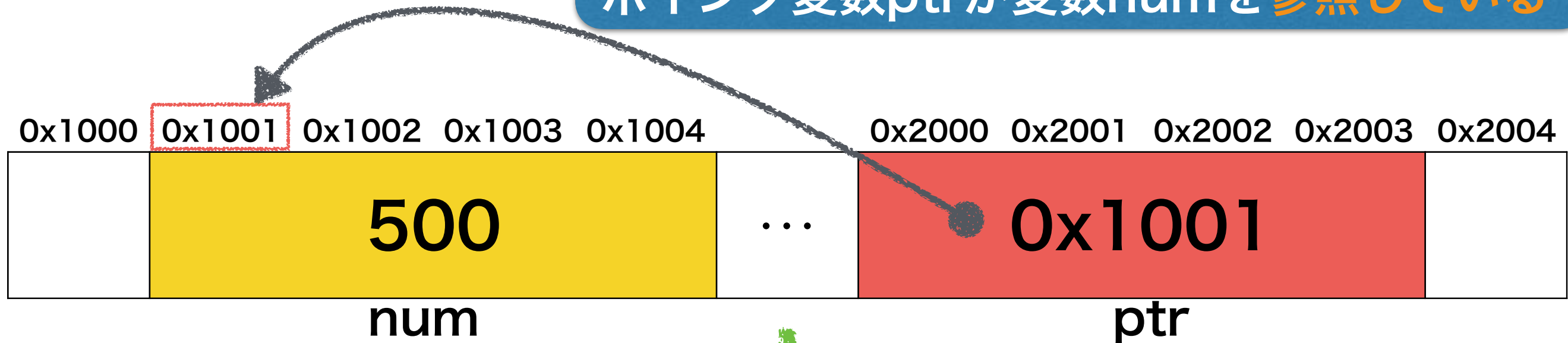
ポインタ変数の前に「\*」（間接参照演算子）を付けると、ポインタが持つアドレスに格納されている値を表す (p.279)



# ポインタのイメージ

「`ptr = &num;`」によって、ポインタ変数`ptr`には変数`num`の先頭アドレスが格納されます。 (p.278)

ポインタ変数`ptr`が変数`num`を参照している



参照は矢印などの線でつないで表すことが多い



# ポインタ・アドレスに関する表記

先程の整数値「500」またはアドレス「0x1001」を表すには、変数numとポインタ変数ptrでは表記が異なります。(p.281)

	整数値 500	アドレス 0x1001
変数num	<b>num</b>	<b><u>&amp;num</u></b>
ポインタ変数ptr	<b><u>*ptr</u></b>	<b>ptr</b>

間接参照演算子

アドレス演算子

The diagram illustrates the notation for variables and their addresses. It features a table with three rows and three columns. The first row serves as a header, distinguishing between integer values and memory addresses. The second and third rows compare the notation for a regular variable 'num' and a pointer variable 'ptr'. For 'num', the variable name is 'num' and its address is '&num'. For 'ptr', the variable name is 'ptr' and the value it holds (the address of the data it points to) is '\*ptr'. Red underlines highlight the '&' and '\*' symbols. Two blue callout boxes at the bottom identify these symbols: '間接参照演算子' (Indirect Reference Operator) for '\*' and 'アドレス演算子' (Address Operator) for '&'.



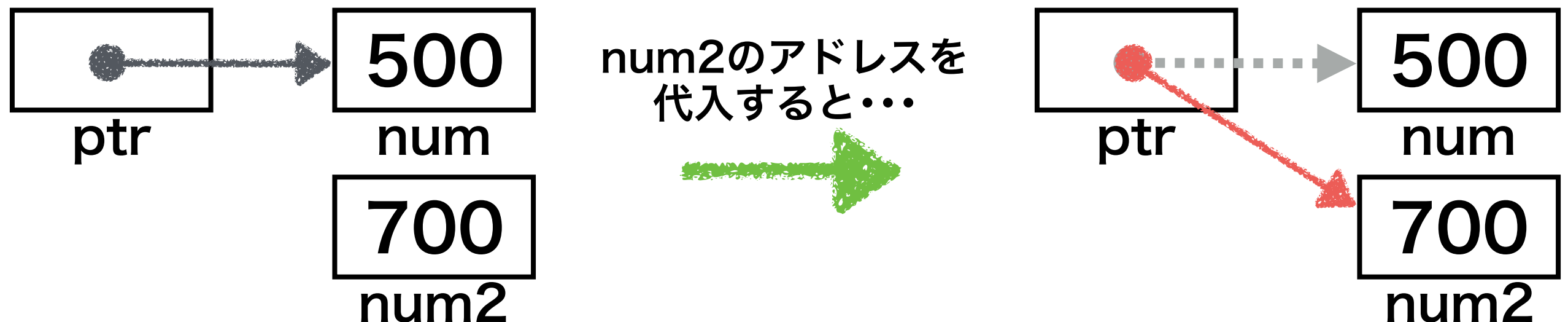
# アドレスの代入

ポインタはアドレスを格納する**変数**であるため、通常の変数と同様に代入処理をすることができます。その場合、ポインタに格納されている値、つまり**アドレス**が**変更**されます。(p.282)

【例3】 (例2の続きに書いたとする)

```
int num2;  
num2 = 700;  
ptr = &num2;
```

変数num2のアドレスを代入すると、**ptrはnum2を参照する**



# ポインタ同士の代入

ポインタからポインタへの代入は、「**アドレスが代入される**」という処理になります。つまり「これらのポインタが**同じ場所を参照する**」ことになります。

【例4】（例3の続きに書いたとする）

```
int *ptr2;  
ptr2 = ptr;
```

ポインタptr2へ、ptrに格納されているアドレスを代入すると、ptr2はptrと**同じ場所を参照する**



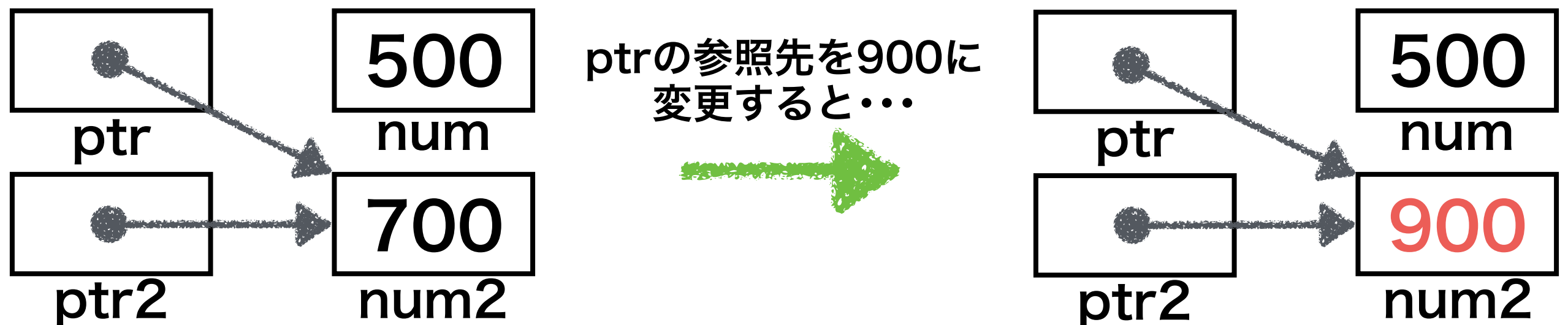
# ポインタで変数の値を変更する

ポインタ演算子を使えば、ポインタが参照している変数の値を変更することができます。 (p.286)

【例5】 (例4の続きに書いたとする)

ptrの参照先 (この時点では変数num2) の値を900に変更する

```
*ptr = 900;  
printf("num1:%d num2:%d ptr:%d \n", num1, num2, *ptr);
```



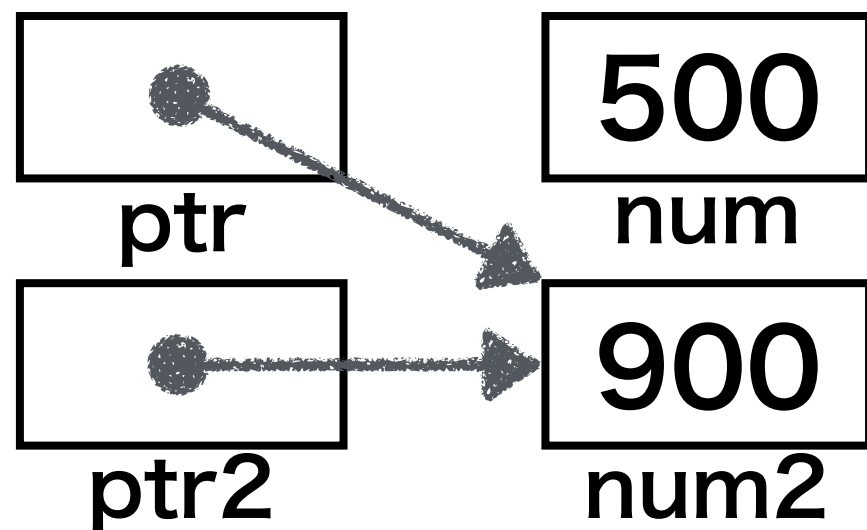
# ポインタの参照を空にする

ポインタに格納されているアドレスを空にする、つまり、何も参照しないようにするには、**NULL**（ナル、ヌル）を代入します。

ptrは何も参照しないようになる  
「ptr = 0」でも可能

【例6】（例5の続きに書いたとする）

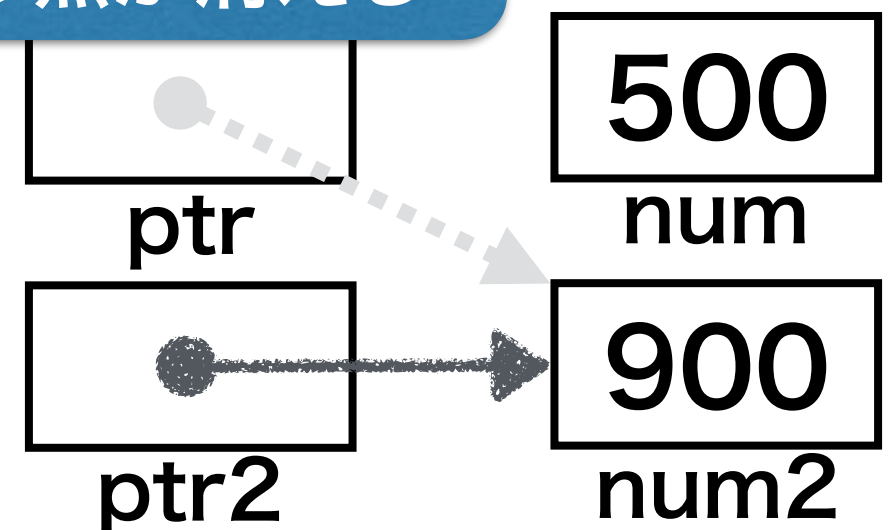
```
ptr = NULL;
```



ptrにNULLを  
代入すると...



参照が消える



# 【練習2-1】

**p.273 Sample1.cとp.279 Sample3.cを入力して  
実行結果を確認しましょう。**

**（もしくは、資料の説明中の例1～例6をmain()に書  
いたプログラムを実行してみましよう。）**

# 【課題2-1】

main()に2つのint型変数を以下のように宣言する。

```
int num2, num1;  
num1 = 100; num2 = 200;
```

この2つの変数に対して、「**変数のアドレス**を比較して、大きい方の**変数のアドレス**とその**変数に格納されている値**を出力する」プログラムを作成してください。

プログラムが完成したら、「**変数を宣言する順番を変えてみて（つまり、int num1, num2）**」実行結果に変化があるか確認してみましょう。

# 【課題2-1】

[ 実行結果 (num2のアドレスの方が大きい場合) ]

大きい方のアドレス: 0x7fff5912c938

そのアドレスに格納されている値: 200

[ 実行結果 (num1のアドレスの方が大きい場合) ]

大きい方のアドレス: 0x7fff5f7d5938

そのアドレスに格納されている値: 100



# 【課題2-2】

main()に、以下のようにポインタptr1を宣言し、num1のアドレスを代入をする。

```
int *ptr1, num1;  
num1 = 1;  
ptr1 = &num1;
```

この処理の後に、「1から9までの**奇数**を出力する」処理を作ってください。ただし、この処理に使える変数は**ポインタptr1のみ**です。（num1も含め他の変数も使わない）

（次のスライドに続きます）

# 【課題2-2（続き）】

「ポインタptr1の参照先の値に対してインクリメント（++）演算をする」場合は、以下のように「間接参照演算子（\*）の処理を優先するため」に括弧を付けると良いです。

```
/* まず間接参照演算子を計算して、インクリメントする */  
/* つまり、「ptr1が参照先の値をインクリメントする」ことになる */  
(*ptr1)++
```

[実行結果]

1 3 5 7 9

# 【課題2-3】

main()に、以下のようにポインタptr2を宣言し、ch1のアドレス代入をする。

```
char *ptr2;  
char ch1 = 'a';  
ptr2 = &ch1;
```

この処理の後に、「**aからzまでの文字**を出力する」処理を作ってください。ただし、この処理に使える変数は**ポインタptr2のみ**です。（ch1も含め他の変数も使わない）

（次のスライドに続きます）

# 【課題2-3（続き）】

「ポインタptr2が参照している値に対してインクリメント（++）演算をする」場合は、課題2-2と同様です。

```
/* まず間接参照演算子を計算して、インクリメントする */  
/* つまり、「ptr2が参照先の値をインクリメントする」ことになる */  
(*ptr2)++
```

[実行結果]

a b c d e f g h i j k l m n o p q r s t u v w x y z

# 【課題2-4】

配列は「同じ型の格納場所をメモリ上に**連続して確保する**」仕組みになっています。main()に、以下のようchar型の配列を宣言します。

```
char array[5];
```

この場合、array[0], array[1], ...のアドレスは、**等間隔**になっています。

これを確認するために、array[0]～array[4]の**アドレスを出力する**プログラムを作ってください。

(次のスライドに続きます)

# 【課題2-4（続き）】

配列の要素array[i]のアドレスは、以下のように表します。

```
&array[i]    /* array[i]に対してアドレス演算子进行处理する */
```

[ 実行結果（char型のサイズは1バイトなので、1バイト間隔になる） ]

0:	0x7fff57c4eaf2	(array[0]のアドレス)
1:	0x7fff57c4eaf3	
2:	0x7fff57c4eaf4	(1バイトずつ増えているのが確認できる)
3:	0x7fff57c4eaf5	
4:	0x7fff57c4eaf6	(array[4]のアドレス)

まだ余裕のある人は…

# 【課題2-5】

課題2-4に対して、char型以外の型についても確認してみましょう。

- ▶ short型 --- 2バイト
- ▶ int型 --- 4バイト
- ▶ long型 --- 8バイト
- ▶ double型 --- 8バイト



# さらに余裕のある人は・・・

**前回作成したスタックのプログラムを拡張して、キューのプログラムも作ってみましょう。（別紙参照）**