

# プログラミングII

<http://bit.ly/Prog3i>

## ソースコードの管理 (2)

前期 第3週

2019/4/23

# 今回は

- ▶バージョン管理とは
- ▶GitHubを使ったバージョン管理
- ▶GitHubを使った基本的な操作方法
- ▶前回のGitHubの準備の続き
- ▶ついでに、**ポインタ**について復習する

# バージョン管理とは

ファイルの変更履歴を記録する

主にプログラム  
(ソースコー  
ド) が対象

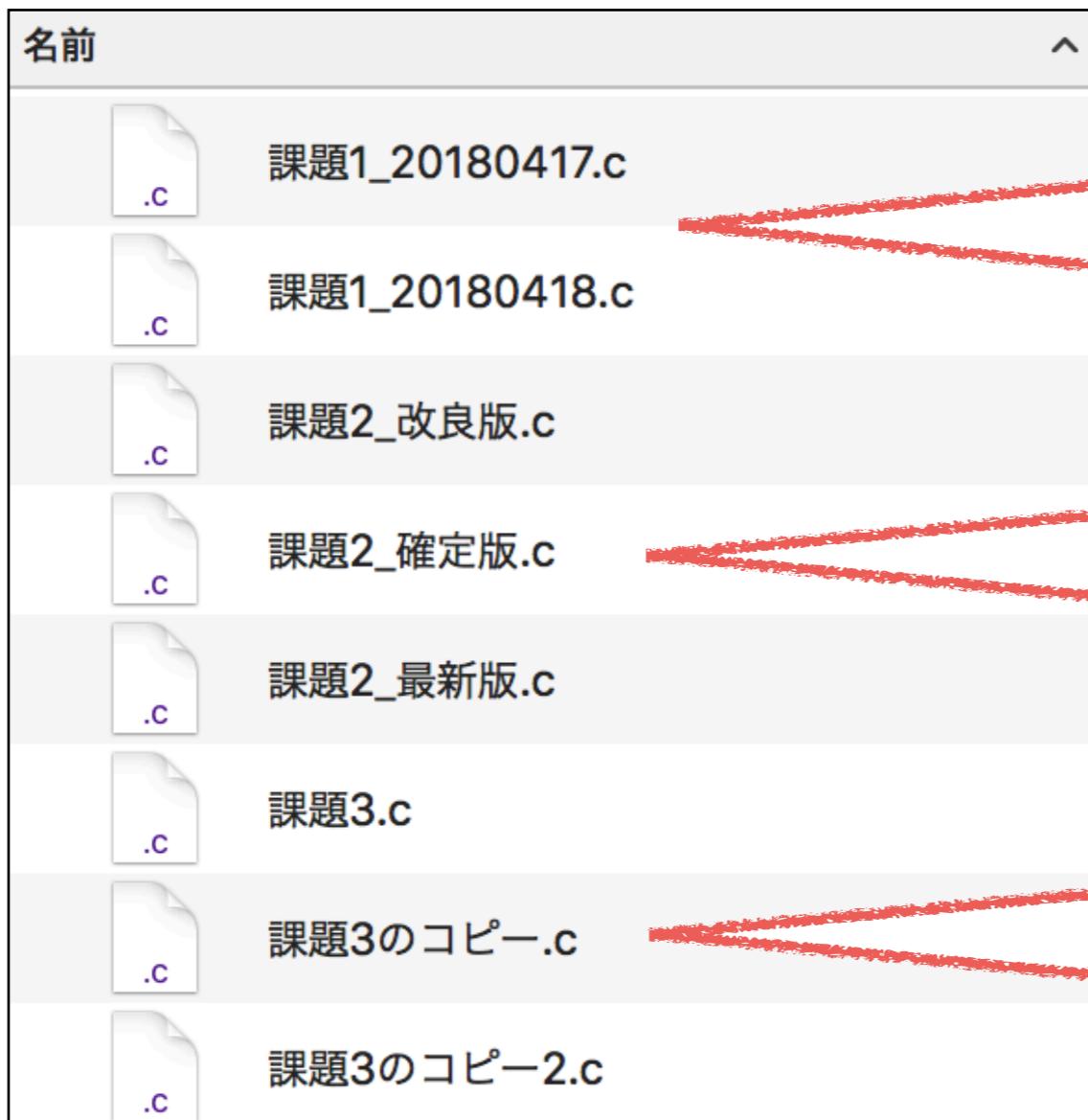
「バージョン」  
と呼ぶ

# バージョン管理の利点

- ▶ プログラムが動かなくなっても、動いていた状態に戻すことができる
- ▶ ソースコードの配付が容易になる
- ▶ 共同開発者との共同開発が容易になる  
→ 現在の商業開発ではバージョン管理システムの導入は必須

# 従来のバージョン管理は・・・

ファイル名で区別するように工夫するが・・・



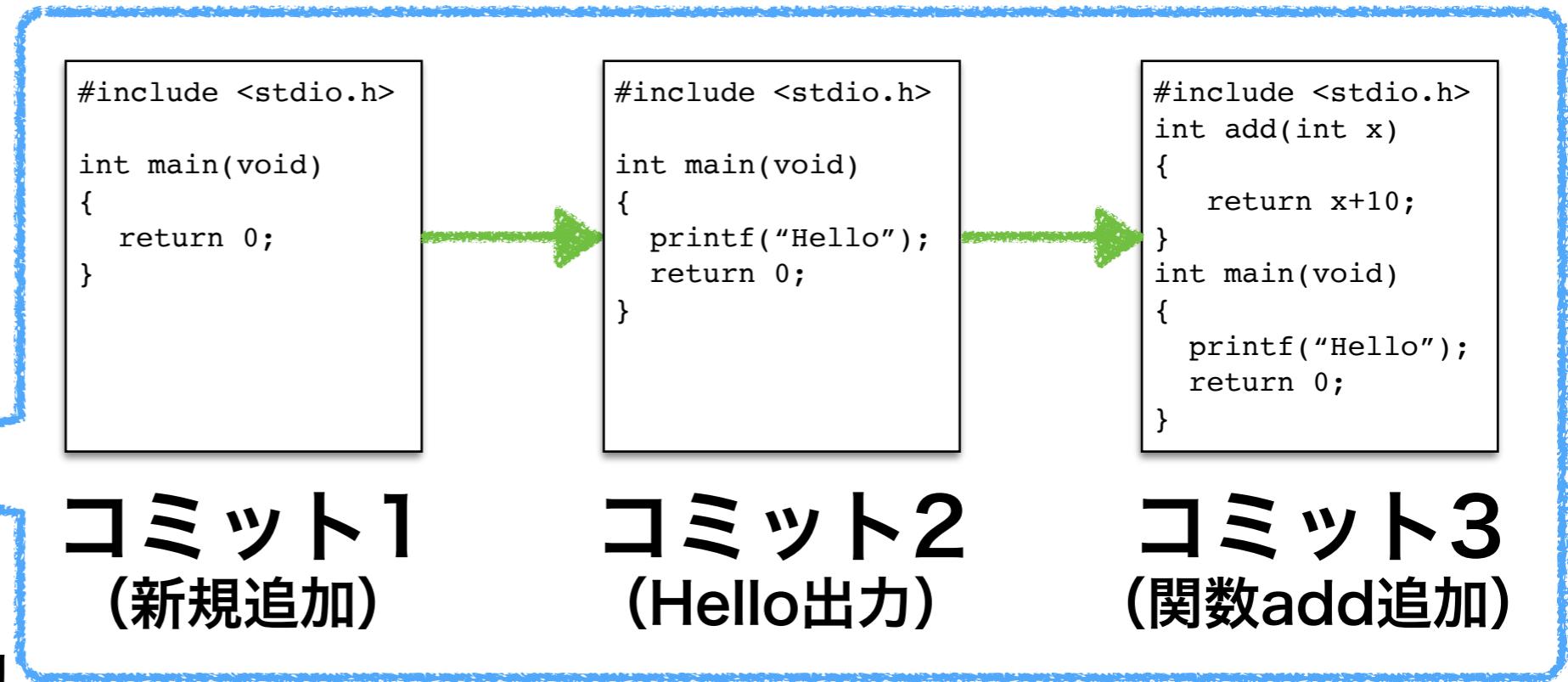
日付を入れて区別できるが、変更内容は不明

結局どれが一番新しいのかが不明

OSが自動で付けるファイル名のまま

# バージョン管理システム

リポジトリに変更履歴を残していく



リポジトリ

(今回は「prog3i-st17?????」)

コミットすると変更内容, 変更時間, コメントなど  
が自動的に記録される

# Gitによるバージョン管理

▶ ステージング…バージョン管理するファイルを指定

◆ 基本コマンド

```
$ git add ファイル名
```

◆ 全てファイルを管理する場合

```
$ git add -A
```

▶ コミット…ステージングしたファイルの変更を記録

◆ 基本コマンド

```
$ git commit -m "コメント"
```

◆ コメントは後から見ても分かるような内容にする

例：課題1完了, 関数addの計算処理を修正

# Gitによるバージョン管理

▶ プッシュ…GitHubにコミット内容を同期

◆基本コマンド (origin masterの意味は次週)

\$ git push origin master

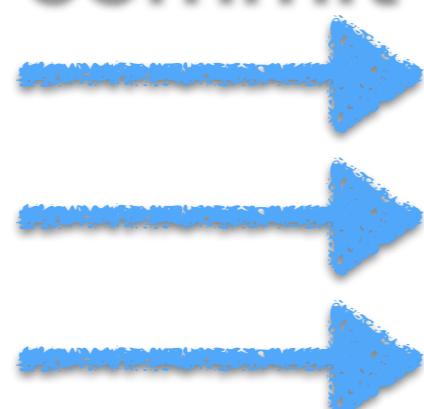
①新規作成時に

add

```
#include <stdio.h>
int add(int x)
{
    return x+10;
}
int main(void)
{
    printf("Hello");
    return 0;
}
```

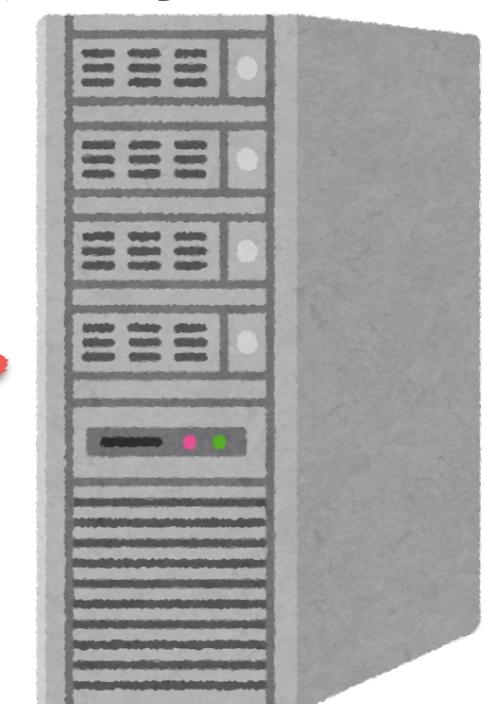
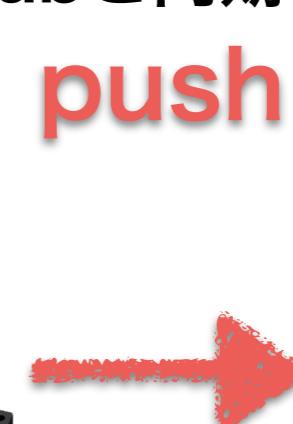
②変更履歴を残す度に

commit



③GitHubと同期する時に

push



リポジトリ

(今回は「prog3i-st17?????」)

GitHub

# 【前回の準備の続き】

今後演習室での手間を少なくできるようにするための準備

端末内で以下のスクリプトを実行して、  
**自分のGitHubリポジトリをPCにダウンロードする**

```
$ ~kogai/mygitclone 「自分のGitHubアカウント名」
```

ちなみに、mygitcloneは以下の3つをまとめて実行しています

```
git config --global http.proxy http://po.cc.ibaraki-ct.ac.jp:3128/  
git config --global https.proxy http://po.cc.ibaraki-ct.ac.jp:3128/  
git clone https://github.com/nit-ibaraki-prog3i/prog3i-$1.git
```

# 【前回の準備の続き】

今後演習室での手間を少なくできるようにするための準備

以下のコマンドを実行して、

自分の名前とメールアドレスの設定をGitHubに入れておく

```
$ cd prog3i-(ユーザ名)  
$ cp ~/kogai/myconf . (←ここにピリオド)  
$ gedit myconf &
```

ファイルを編集して、自分の名前とメールアドレスを書く

```
#!/bin/sh  
git config --global user.name "自分の名前（ローマ字）"  
git config --global user.email "学校のGmailアドレス@gm.ibaraki-ct.ac.jp"
```

# 【前回の準備の続き】

今後演習室での手間を少なくできるようにするための準備

以下のコマンドを実行して、

今回の変更をGitHubにアップロードしておく

1. 端末内で、

```
$ ./myconf  
$ git add -A  
$ git commit -m "myconf追加"  
$ git push origin master
```

2. 以下の自分の課題用リポジトリを開いて、`myconf`がアップロードされていることを確認する

[https://github.com/nit-ibaraki-prog3i/prog3i-\(ユーザ名\)](https://github.com/nit-ibaraki-prog3i/prog3i-(ユーザ名))

# 次回からは…

演習室で作業する前に、以下のコマンドを入れるだけで準備が完了する

```
$ ~kogai/mygitclone 「自分のGitHubアカウント名」  
$ cd prog3i-(ユーザ名)  
$ ./myconf
```

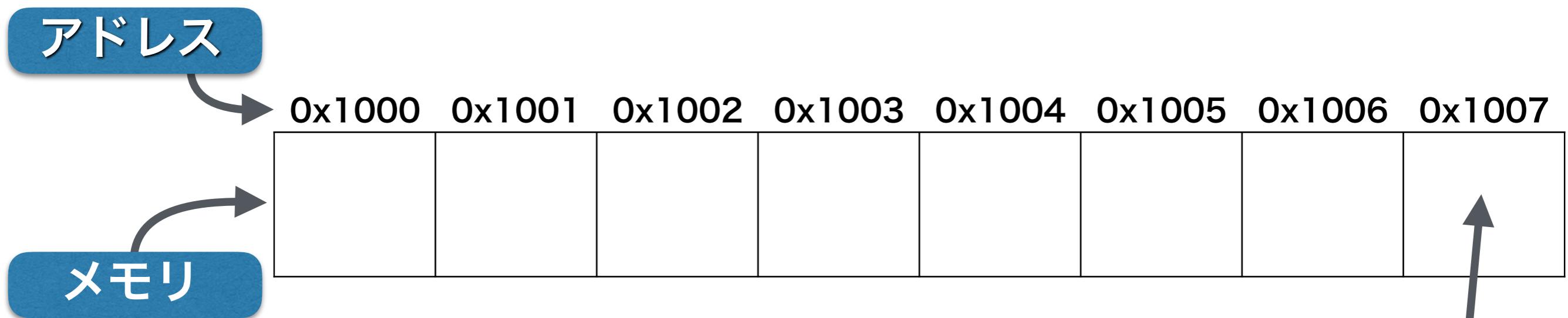
※本体をシャットダウンするまでは、  
上記「mygitclone」と「myconf」の設定は有効です

# 今回の課題は…

- ▶自分で作ったプログラムファイルをGitHubでバージョン管理する方法を把握する
- ▶ついでに、**ポインタ**について復習する  
(以降に、D科2年生プログラミングでの資料を載せます。説明文中のページは、やさしいCのページ番号になります。)

# 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。



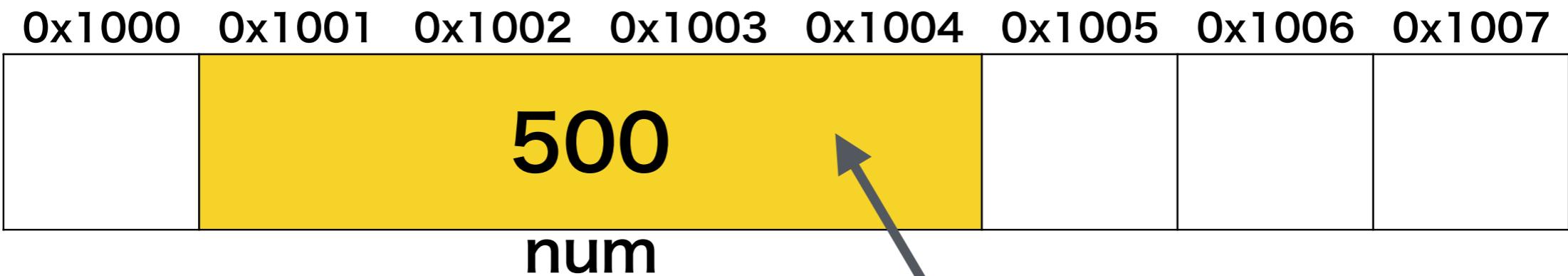
アドレスは順番通りにメモリに割り振られている  
(p.272)

# 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1】

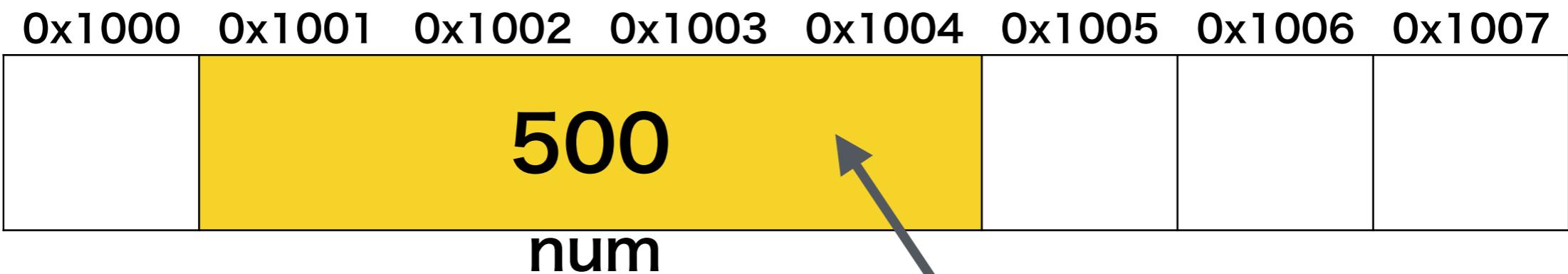
```
int num;  
num = 500;
```



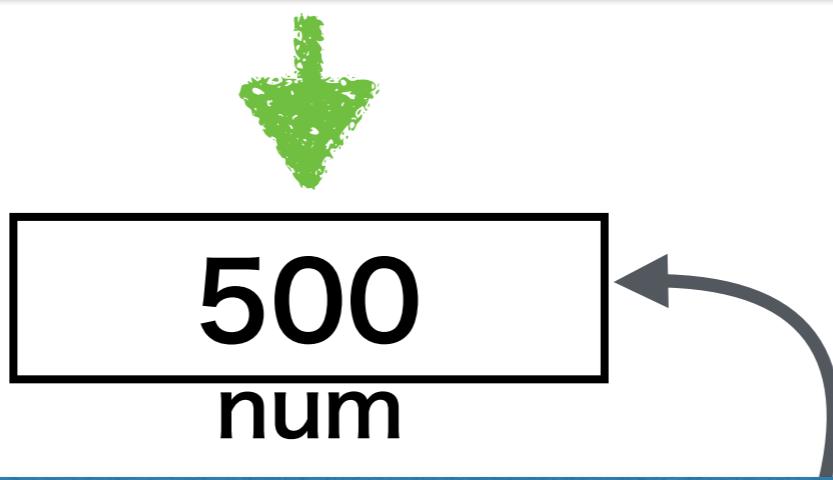
0x1001～1004というアドレスに格納場所（int型なので4バイト分）が確保された。メモリ上で空いている場所に確保されるため、実際には確保される場所は実行するたびに変化する。

# 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…



**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

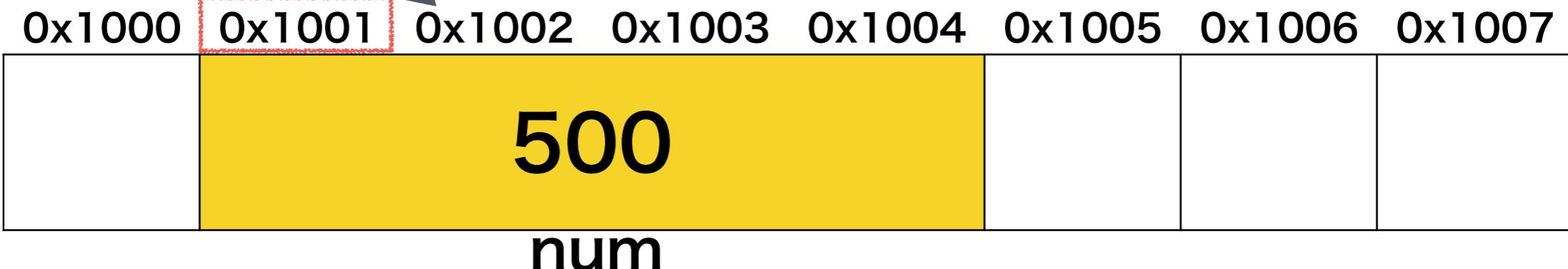
変数の格納場所のアドレスを、扱うには「**&変数名**」のようにアドレス演算子「**&**」を使います。 (p.273)

【例2】

```
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3】

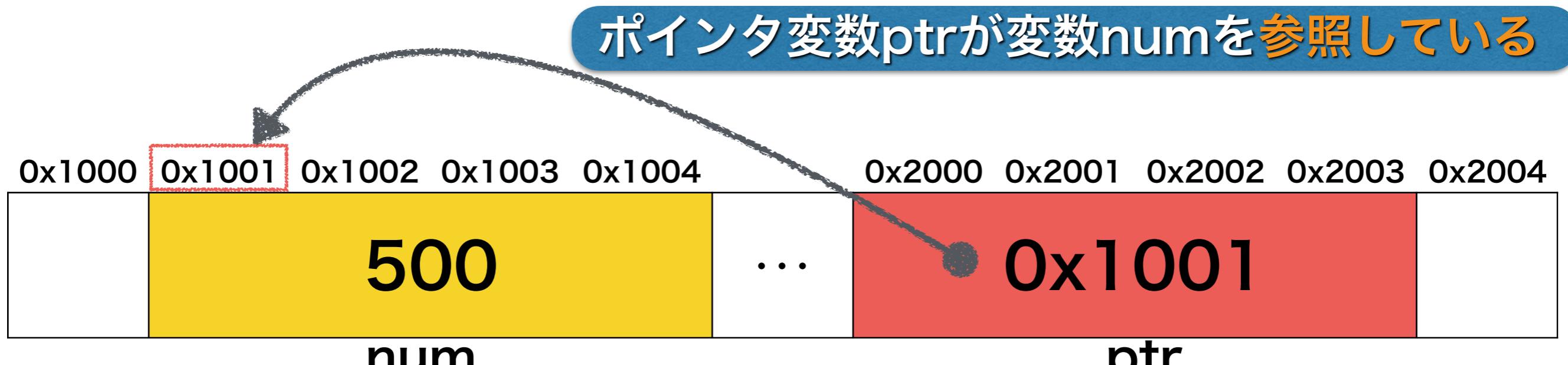
```
int num;  
int *ptr;  
num = 500;  
ptr = &num;  
printf("num:%d ptr:%d \n", num, *ptr);
```

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)

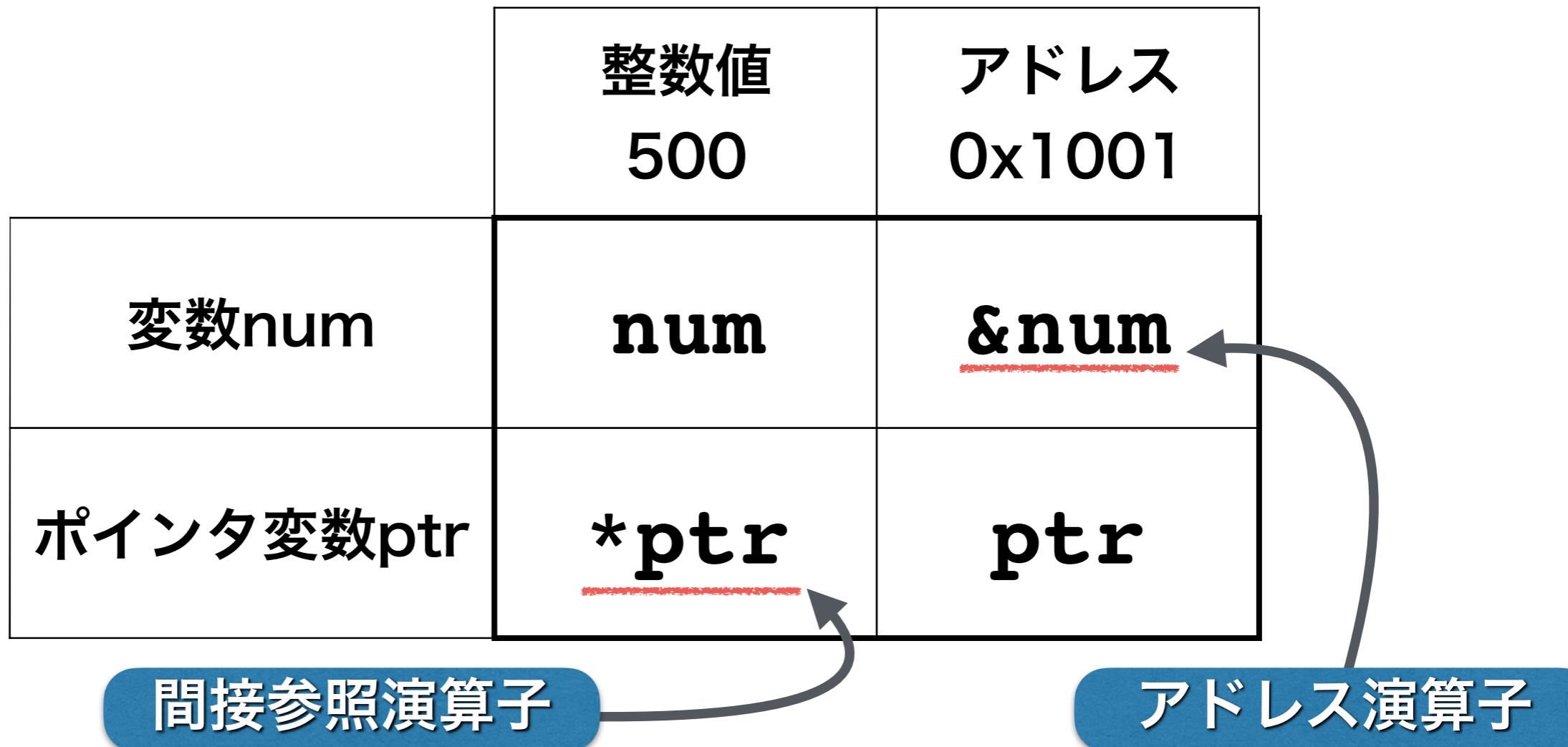


参照は矢印などの線でつないで表すことが多い



# 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)



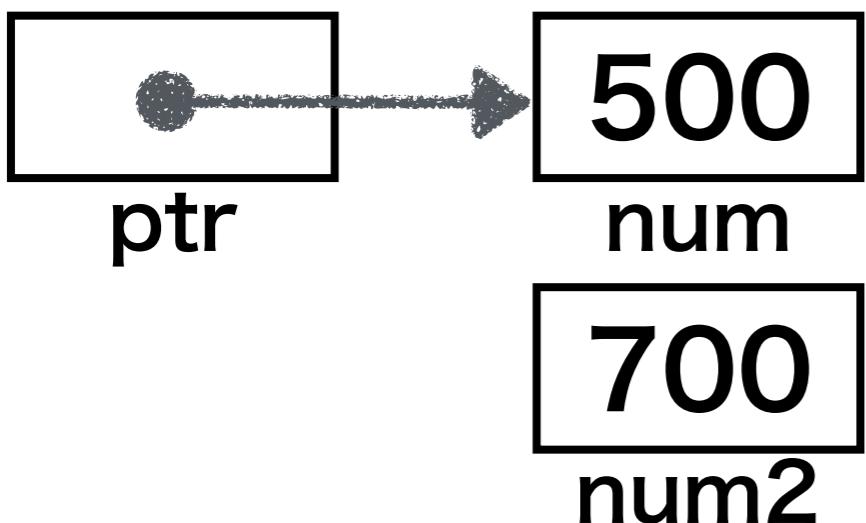
# アドレスの代入

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)

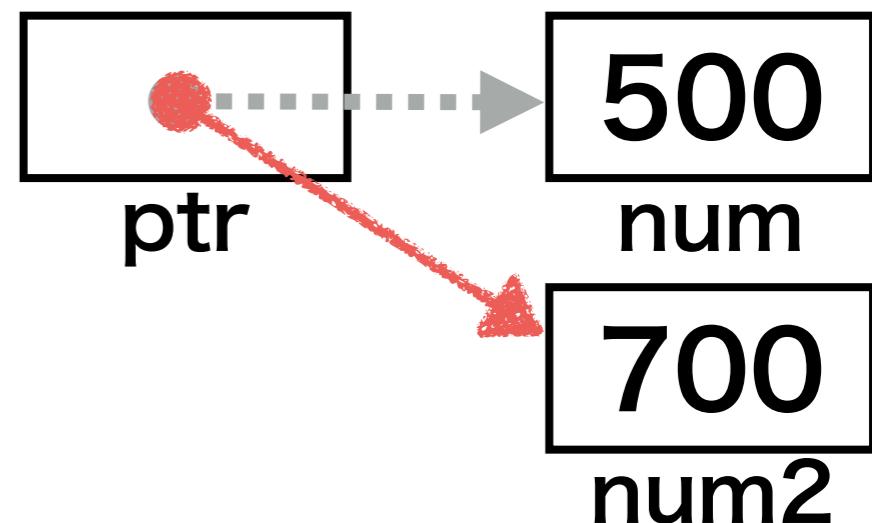
【例4】(例3の続きを書いたとする)

```
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



num2のアドレスを  
代入すると…



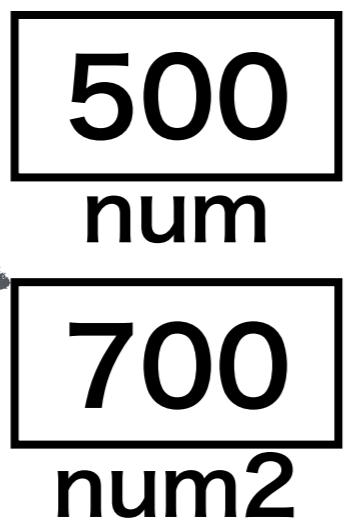
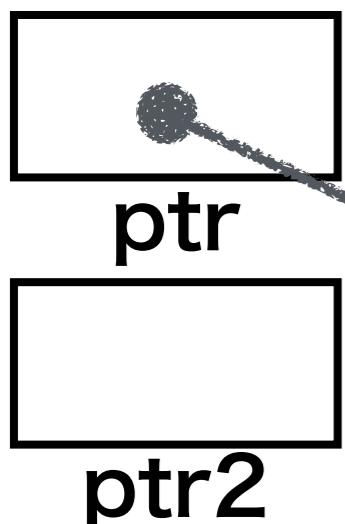
# ポインタ同士の代入

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。

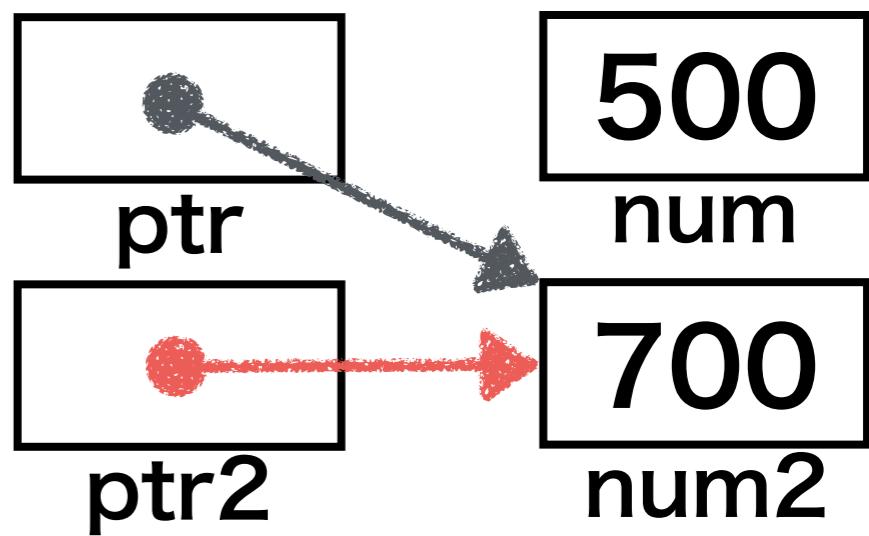
【例5】(例4の続きを書いたとする)

```
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



ptrをptr2へ  
代入すると…  
→



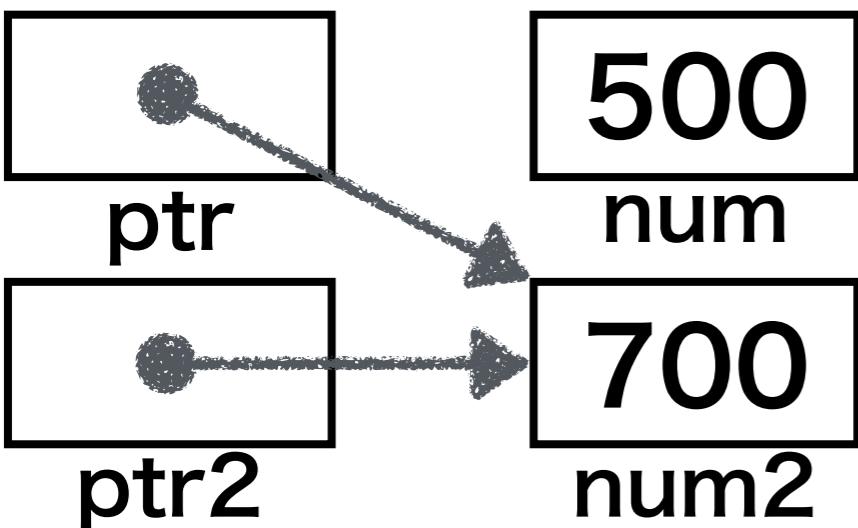
# ポインタで変数の値を変更する

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)

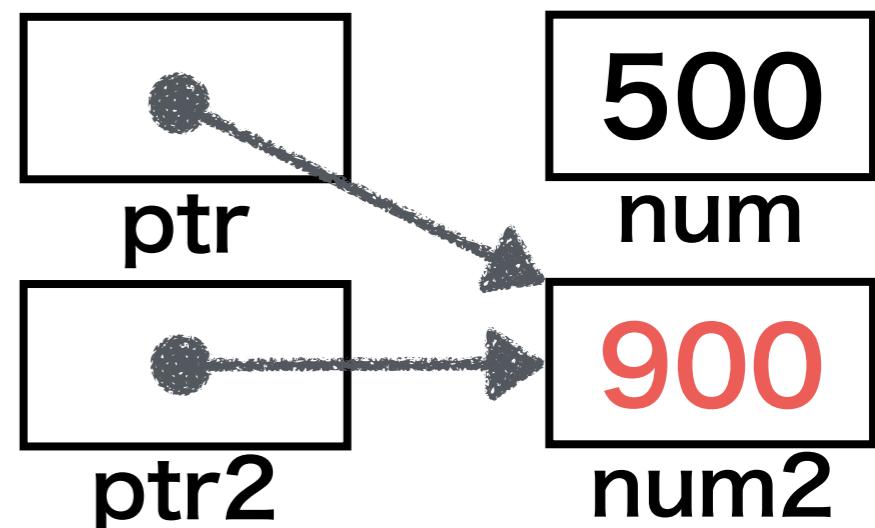
## 【例6】(例5の続きに書いたとする)

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("num:%d num2:%d ptr:%d \n", num, num2, *ptr);
```



ptrの参照先を900に  
変更すると…



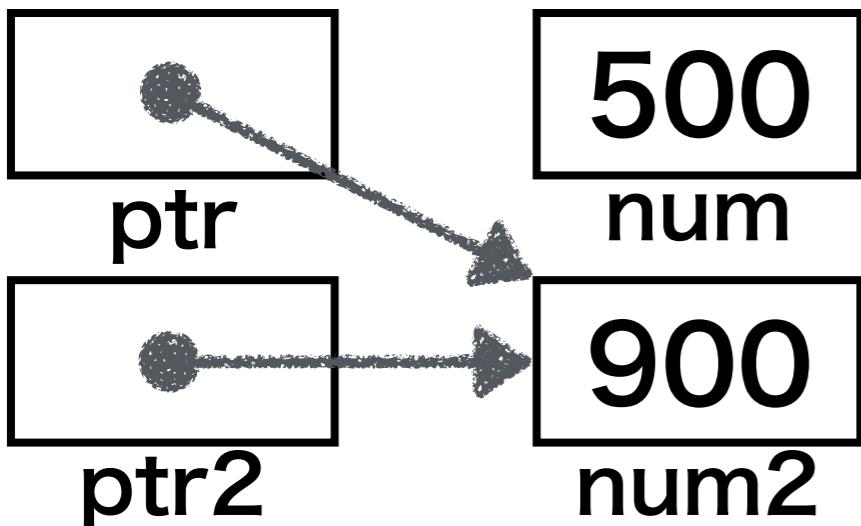
# ポインタの参照を空にする

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

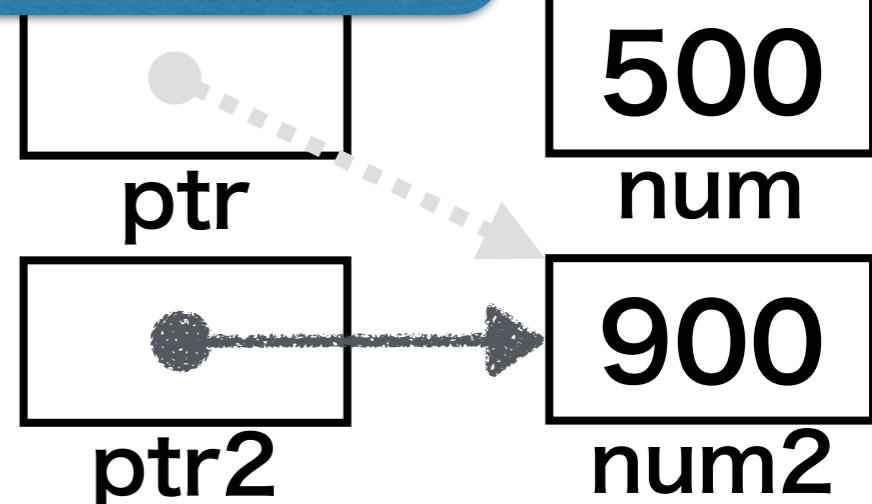
【例7】（例6の続きに書いたとする）

`ptr = NULL;`



ptrにNULLを  
代入すると…

参照が消える



# 【課題の準備】

以下の流れで、課題のプログラムを作るためのフォルダを準備しましょう。

1. 端末を起動して、以下のコマンドを実行して前期第3週のフォルダを作る

```
$ cd prog3i-(ユーザ名)
```

```
$ mkdir week103
```

```
$ cd week103
```

2. テキストエディタで、「example.c」というファイルを新規に開く

```
$ gedit example.c &
```

(geditは&を付けて実行すると端末で続きの作業ができる)

# 【課題3-1】

資料中の例3をmain()に書いたプログラムを作って、  
実行してみましょう。

# 【課題の提出】

以下の流れで、作ったCプログラムをGitHubにプッシュして、Webサイトで確認してみましょう。

1. 端末内で、以下のコマンドで課題を提出

```
$ git add -A  
$ git commit -m "課題3-1提出"  
$ git push origin master
```

2. 自分の課題用リポジトリを開いて、week103がプッシュされているのを確認する  
(以下の場所は、github.comを開いてログインしてもリンクが現れる)

[https://github.com/nit-ibaraki-prog3i/prog3i-\(ユーザ名\)](https://github.com/nit-ibaraki-prog3i/prog3i-(ユーザ名))

# 【課題3-2】

課題3-1で作成したexample.cの続きを、資料中の例4～例6をmain()に追加したプログラムを作って、実行してみましょう。

GitHubにプッシュして、example.cの変更履歴を確認してください。（手順は、以降のスライド参照）

# 【課題の提出】

以下の流れで、作ったCプログラムをGitHubにプッシュして、Webサイトで確認してみましょう。

1. 端末内で、以下のコマンドで課題を提出

```
$ git add -A  
$ git commit -m "課題3-2提出"  
$ git push origin master
```

2. 自分の課題用リポジトリを開いて、example.cの変更履歴を確認する  
(次のスライドにある画面参考)

[https://github.com/nit-ibaraki-prog3i/prog3i-\(ユーザ名\)](https://github.com/nit-ibaraki-prog3i/prog3i-(ユーザ名))

# 【課題3-2】 の確認方法

自分のリポジトリでコミット内容を確認します

クリック

5 commits      1 branch      0 releases      1 contributor

Branch: master ▾      New pull request      Create new file      Upload files      Find file      Clone or download ▾

kogailab	配列の要素変更3	Latest commit 17ad131 2 minutes ago
week01	配列の要素変更3	2 minutes ago
README.md	はじめての課題退出	15 hours ago

※画面は一例です

# 【課題3-2】 の確認方法

自分のリポジトリでコミット内容を確認します

クリック

配列の要素変更  
kogailab committed 28 minutes ago

第1週練習提出  
kogailab committed 38 minutes ago

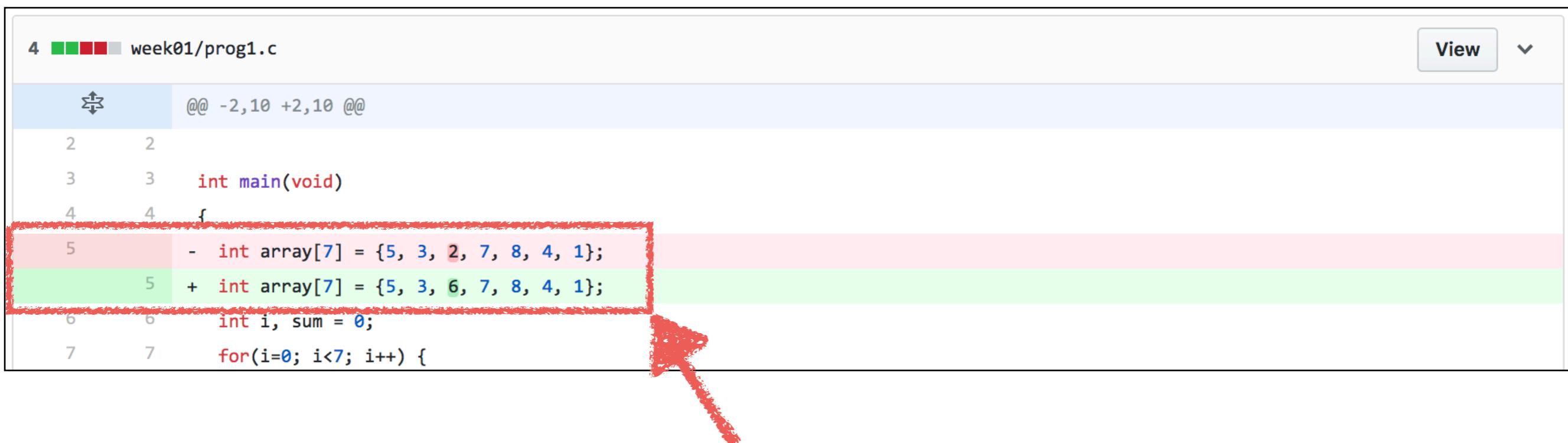
Commits on Apr 16, 2018

はじめての課題退出  
kei534 committed 15 hours ago

※画面は一例です

# 【課題3-2】 の確認方法

自分のリポジトリでコミット内容を確認します



4 week01/prog1.c View ▾

```
diff --git a/week01/prog1.c b/week01/prog1.c
@@ -2,10 +2,10 @@
2   2
3   3     int main(void)
4   4 {
5 -   int array[7] = {5, 3, 2, 7, 8, 4, 1};
5 +   int array[7] = {5, 3, 6, 7, 8, 4, 1};
6   6     int i, sum = 0;
7   7     for(i=0; i<7; i++) {
```

5行目の変更前 (-) と変更後 (+) の内容が確認できる

※画面は一例です

# 【課題3-3】

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処理を、ファイル  
「kadai.c」を作成します。

```
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3-3（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

# 【課題の提出】

以下の流れで、作ったCプログラムをGitHubにプッシュして、Webサイトで確認してみましょう。

1. 端末内で、以下のコマンドで課題を提出

```
$ git add -A  
$ git commit -m "課題3-3提出"  
$ git push origin master
```

2. 自分の課題用リポジトリを開いて、`kadai.c`の変更履歴を確認する

[https://github.com/nit-ibaraki-prog3i/prog3i-\(ユーザ名\)](https://github.com/nit-ibaraki-prog3i/prog3i-(ユーザ名))

# 【課題3-4】

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処理を、ファイル「kadai.c」の課題3-3の続きに書きます。

```
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3-4（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（課題3-3の結果に続けて以下が outputされる） ]

```
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
```

# 【課題の提出】

以下の流れで、作ったCプログラムをGitHubにプッシュして、Webサイトで確認してみましょう。

1. 端末内で、以下のコマンドで課題を提出

```
$ git add -A  
$ git commit -m "課題3-4提出"  
$ git push origin master
```

2. 自分の課題用リポジトリを開いて、`kadai.c`の変更履歴を確認する

[https://github.com/nit-ibaraki-prog3i/prog3i-\(ユーザ名\)](https://github.com/nit-ibaraki-prog3i/prog3i-(ユーザ名))

まだ余裕のある人は…

## 【課題3-5】

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、以下のように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。ファイル「kadai2.c」に作成してください。

(次のスライドに続きます)

# 【課題3-5（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を処理する */
```

[ 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)

# 【課題の提出】

以下の流れで、作ったCプログラムをGitHubにプッシュして、Webサイトで確認してみましょう。

1. 端末内で、以下のコマンドで課題を提出

```
$ git add -A  
$ git commit -m "課題3-5提出"  
$ git push origin master
```

2. 自分の課題用リポジトリを開いて、`kadai2.c`の変更履歴を確認する

[https://github.com/nit-ibaraki-prog3i/prog3i-\(ユーザ名\)](https://github.com/nit-ibaraki-prog3i/prog3i-(ユーザ名))