STAGE 1-5

関数を使う

「関数」のプログラミングにおける再定義

🚩 このSTAGEの目標

- プログラミングにおける「関数」を知る
- 関数の書き方がわかる

かん-すう

関数

function

数値やデータを受け取って
処理を行うもの、または処理のまとまりをさす

必ずしも値を出力するわけではないし
1つの入力に対する出力は一意である必要はない

関数 かん-すう

全ての関数は、
「戻り値の型」「名前」を持つ
また、「引数」(ひきすう) が与えられる場合もある

引数 は 関数の入力
戻り値 は 関数の出力

ところが
引数も戻り値も、どちらも
無いことがある

$$\sin(\pi) = 0$$

入力(引数)　出力(戻り値)

関数 かん-すう

全ての関数は、
「戻り値の型」「名前」を持つ
また、「引数」(ひきすう) が与えられる場合もある

戻り値が不要な関数の呼び出し方 (実行のしかた) は

```
printf( "%d", 3 );
```

関数の名前 (引数1, 引数2, ...,)

出力
```
3
```

引数はカンマで区切り、順番を変えてはいけない
引数には定数のほかに変数を書くこともできる

関数 かん-すう

全ての関数は、
「戻り値の型」「名前」を持つ
また、「引数」(ひきすう) が与えられる場合もある

戻り値のある関数の呼び出し方 (実行のしかた) は

```
a = pow( 2, 4 );
```

$a = 2^4$

変数 = 関数の名前 (引数1, 引数2, ...,)

変数の型 は 関数の戻り値の型 と一致している必要がある

関数 かん-すう

全ての関数は、
「戻り値の型」「名前」を持つ
また、「引数」(ひきすう) が与えられる場合もある

戻り値のある関数は、定数のように書いてよい

```
printf( “%f”, pow(2, 4) );
```

printf 関数の引数は “%f” と pow(2, 4)
pow 関数の引数は 2 と 4

出力
```
16
```

定数のように書いた関数の戻り値の型 は
その関数の引数の型 と一致している必要がある

関数 かん-すう

★ printf 関数の使い方　戻り値 `int` 表示された文字数、エラー時は負の数

引数1 `文字列` 表示したい文字列 (“” で囲う)
ただし、文字列の中の %d (整数) や %lf (小数) 、%s (文字列) … などの %<文字> で表されるやつは、引数2,3,4 … に置き換えられます
ただの「%」を出すには %% を入れます

※ 変数名を “” の中に入れるだけでは、変数の値は表示されない

引数n n 番目の引数は、引数1 のうちの n-1 番目のフォーマット文字列を置き換えます

※ n: 2以上の整数. 定数を書いてもいいです

コード

```
int i = 3;
double d = 12.34;
printf( “int=%d, double=%lf%%, str=%s”, i, d, “abcdefg” );
```

出力

```
int=3, double=12.34%, str=abcdefg
```

関数 かん-すう

★ printf 関数の使い方

引数1 表示したい文字列 (“” で囲う)

文字列

また、この文字列の中に ¥n を追加することで、文字列の途中でも改行できます
文字列に ¥n を入れない場合、最後に表示された文字の直後につなげて表示されます

コード

```
int i = 3;
double d = 12.34;
printf( “i is %d¥n”, i );
printf( “d is %lf¥n”, d );
```

出力

```
i is 3
d is 12.34
```

🚩 この STAGE の目標

- ☑ プログラミングにおける「関数」を知る
- ☑ 関数の書き方がわかる

できるようになったこと

自作の関数が書ける

関数 かん-すう

全ての関数は、
「戻り値の型」「名前」を持つ
また、「引数」(ひきすう) が与えられる場合もある

標準ライブラリに定義されている関数では
どうしても足りない場合、
新しく関数を作ることができる

「関数を**定義**する」という

関数 かん-すう

全ての関数は、
「戻り値の型」「名前」を持つ
また、「引数」(ひきすう) が与えられる場合もある

関数は**定義**することではじめて使うことができる

```
int twiceIt(int value) {     // 戻り値の型 名前 (引数1の型 引数1の名前) {
  int result = value * 2;    // 処理を書く
  return result;             //  戻り値をどこかで返す
}                            // } で「ブロック」を閉じる
```

戻り値のある関数を定義する例

関数 かん-すう

全ての関数は、
「戻り値の型」「名前」を持つ
また、「引数」(ひきすう) が与えられる場合もある

関数は**定義**することではじめて使うことができる

```
void printIt(int value) {    // 戻り値の型 名前 (引数1の型 引数1の名前) {
  printf(value);             // 処理を書く
  return;                    // void 型の関数は戻り値は返さない
}                            // } で「ブロック」を閉じる
```

戻り値のない関数を定義する例

関数 かん-すう

```
void printIt(int value) {    // 戻り値の型 名前 (引数1の型 引数1の名前) {
  printf(value);             // 処理を書く
  return;                    //  void 型の関数は戻り値は返さない
}                            // } で「ブロック」を閉じる
```

void 型

関数にだけ使える、
「関数に戻り値がない」ことを示す型

引数はあってもなくてもいい

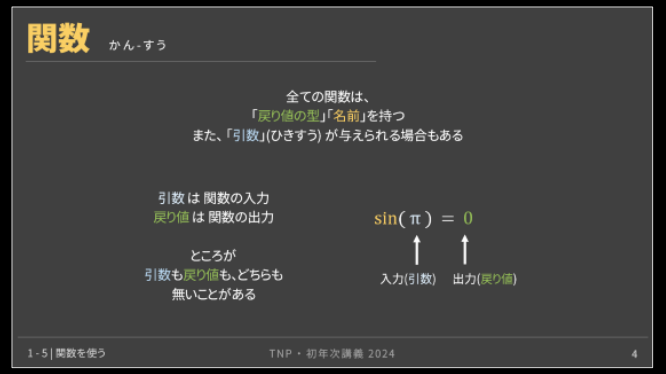
void: 虚無 (転じて「無を返す」関数)

練習

与えられた整数に 7 を足した数を返し、
ついでにその計算式を表示する関数 add_seven
を作ってみよう

引数 | 引数1 `int` 足される数

戻り値 | `int` 引数1 に 7 を足した数



例

コード

```
int a = add_seven( 4 );
printf( "a=%d", a );
```

出 力

```
4+7=11
a=11
```

引数1 4 に 7 を足した数は 11 なので、add_seven 関数は “4+7=11” を出力し
戻り値 11 を返します
a が初期化されて 11 が入るので、2行目は “a=“のあとに a の値を出力します

練習

与えられた整数に７を足した数を返し、
ついでにその計算式を表示する関数
`add_seven` を作ってみよう

引数 | `引数1` `int` 足される数

戻り値 | `int` 引数1 に７を足した数

コード

```
int add_seven( int plusWhat ) {

  // さて、なにを書こう？

}
```

例

コード

```
int a = add_seven(4)
printf( “a=%d”, a )
```

出 力

```
4+7=11
a=11
```

`plusWhat` に４が代入されている状態で、関数 `add_seven` の中身(波かっこの中のコード) が実行される
`add_seven` は `(plusWhat)+7=(plusWhatに7を足した数)` を `printf()` で出力しつつ、
`return (plusWhatの値に7を足した数)` で値を返す

練習

与えられた整数に 7 を足した数を返し、ついでにその計算式を表示する関数 `add_seven` を作ってみよう

引数 | 引数1 `int` 足される数

戻り値 | `int` 引数1 に 7 を足した数

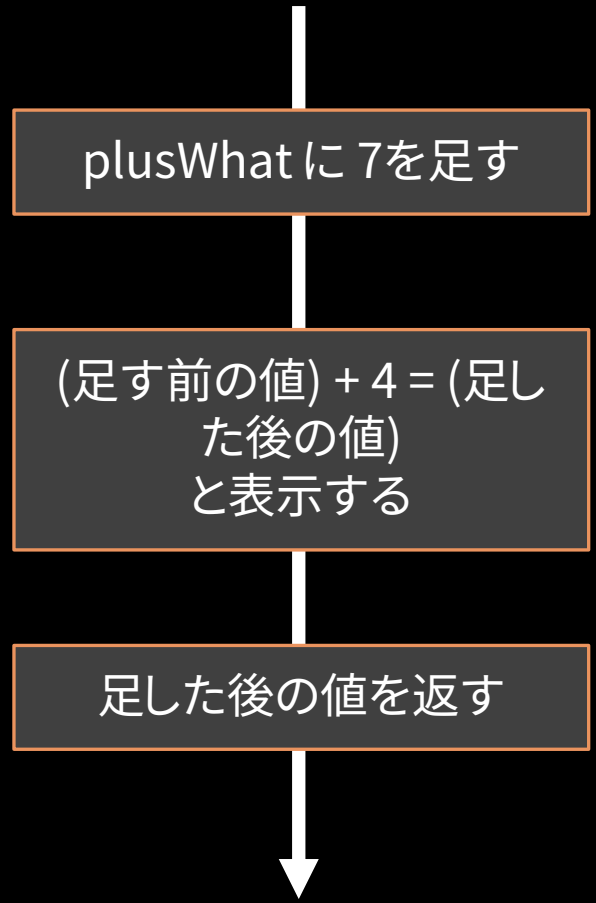
コード

```
int add_seven( int plusWhat ) {

  // さて、なにを書こう？

}
```

フローチャート



例

コード

```
int a = add_seven(4)
printf( "a=%d", a )
```

出力

```
4+7=11
a=11
```

練習

与えられた整数に 7 を足した数を返し、ついでにその計算式を表示する関数 `add_seven` を作ってみよう

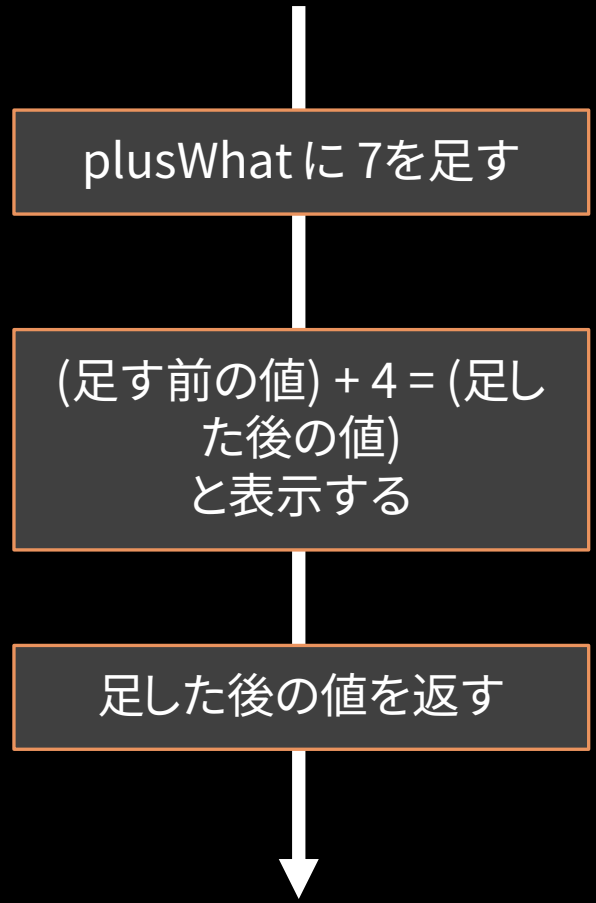
引数 | 引数1 `int` 足される数

戻り値 | `int` 引数1 に 7 を足した数

コード

```
int add_seven( int plusWhat ) {
  plusWhat = plusWhat + 7;
  printf( "%d+4=%d", 足す前, plusWhat);
  return plusWhat;
}
```

フローチャート



例

コード

```
int a = add_seven(4)
printf( "a=%d", a )
```

出力

```
4+7=11
a=11
```

練習

与えられた整数に 7 を足した数を返し、ついでにその計算式を表示する関数 `add_seven` を作ってみよう

引数 | 引数1 `int` 足される数

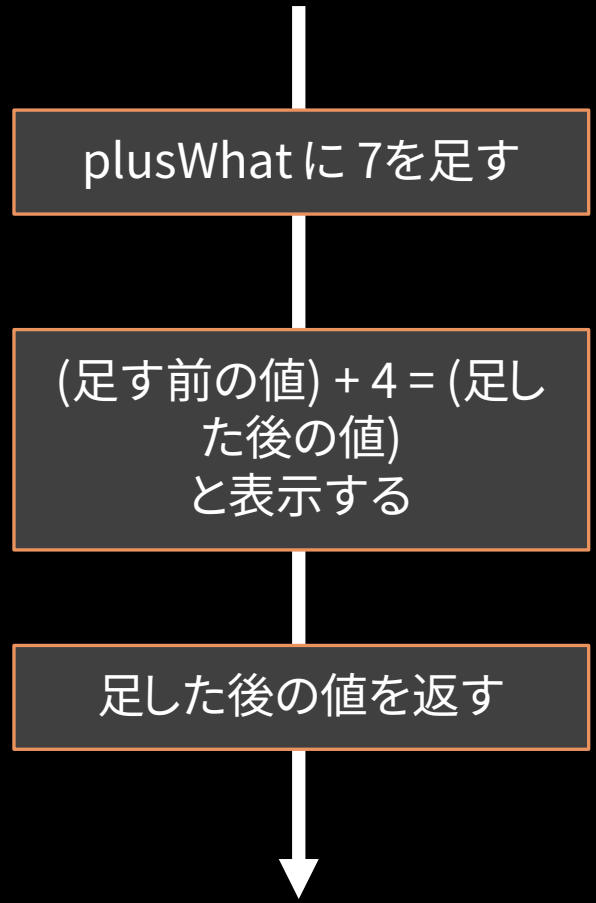
戻り値 | `int` 引数1 に 7 を足した数

コード

```
int add_seven( int plusWhat ) {
  plusWhat = plusWhat + 7; // 足す前の値が上書きされた
  printf( "%d+4=%d", 足す前, plusWhat);
  return plusWhat;
}
```

足す前の値が変数の中にない　ので (足した後 - 7) をわざわざ書く?

フローチャート



例

コード

```
int a = add_seven(4)
printf( "a=%d", a )
```

出力

```
4+7=11
a=11
```

練習

与えられた整数に 7 を足した数を返し、ついでにその計算式を表示する関数 add_seven を作ってみよう

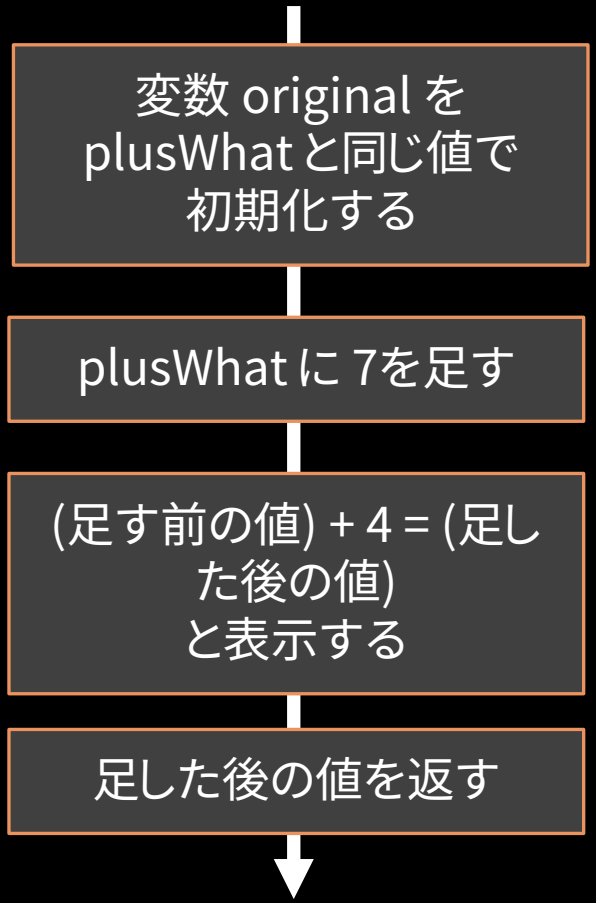
引数 | 引数1 int 足される数

戻り値 | int 引数1 に 7 を足した数

コード

```
int add_seven( int plusWhat ) {
  int original = plusWhat; // original に元の値を入れる
  plusWhat = plusWhat + 7; // 足す前の値を上書き
  printf( "%d+4=%d", original, plusWhat);
  return original;
}
```

フローチャート



例

コード

```
int a = add_seven(4)
printf( "a=%d", a )
```

出力

```
4+7=11
a=11
```

練習

与えられた整数に 7 を足した数を返し、
ついでにその計算式を表示する関数
`add_seven` を作ってみよう

引数 | 引数1 `int` 足される数

戻り値 | `int` 引数1 に 7 を足した数

コード

```
int add_seven( int plusWhat ) {
  int original = plusWhat; // original に元の値を入れる
  plusWhat = plusWhat + 7; // 足す前の値を上書き
  printf( "%d+4=%d", original, plusWhat);
  return plusWhat;
}
```

これで目標が達成できた

実行するときは、このコードを main 関数の外に設置して、
main 関数に「例」のコードを書こう

練習

与えられた整数に 7 を足した数を返し、
ついでにその計算式を表示する関数
`add_seven` を作ってみよう

引数 | 引数1 `int` 足される数

戻り値 | `int` 引数1 に 7 を足した数

コード

```
int add_seven( int plusWhat ) {
  int original = plusWhat; // original に元の値を入れる
  plusWhat = plusWhat + 7; // 足す前の値を上書き
  printf( “%d+4=%d”, original, plusWhat);
  return original;
}

int main(void) { // main 関数も例示するとこんな感じ
  int a = add_seven(4);
  printf( “a=%d”, a );
}
```

関数　かん-すう

★ printf 関数の使い方　戻り値 `int` 表示された文字数、エラー時は負の数

引数1 `*char[]` `string`

表示したい文字列 (“” で囲う)
ただし、文字列の中の %d (int型) や %f (float), %lf (double型) 、%s (文字列) … などの %<1文字> で表される「フォーマット指定子」は、引数2,3,4 … に置き換えられます
%1$d のように %n$d と %番号$型 で表される「順序指定フォーマット指定子」を入れると、文字列の後からn番目の引数に置き換えられます (% の他に アスタリスク * も使えます)
%<桁数><型:d¦f¦lf> とすると桁数まで半角スペースで埋めます (%5d)
%0<桁数><型:d¦f¦lf> とすると桁数までゼロ埋めされます (%05d)
ただの「%」を出すには %% を入れます
※ 他の型も表示できるので必要なら検索してみよう

引数n `Any`

n 番目の引数は、引数1 のうちの n-1 番目のフォーマット文字列を置き換えます
※ n: 2以上の整数. 定数を書いてもいいです

練習

与えられた整数に７を足した数を返し、ついでにその計算式を表示する関数を作ってみよう

引数 | 引数1 `int` 足される数

戻り値 | `int` 引数1に７を足した数

> %1$d のように %n$d と %番号$型 で表される「順序指定フォーマット指定子」を入れると、文字列の後からn番目の引数に置き換えられます

コード

```
int add_seven( int plusWhat ) {
  int result = plusWhat + 7;
  printf( "%1$d+%2$d=%3$d", plusWhat, 7, a );
  return result;
}
```

複数回にわたって同じ内容を表示する際に強力

練習

%<桁数><型:d¦f¦lf>とすると桁数まで半角スペースで埋めます (%5d)
%0<桁数><型:d¦f¦lf>とすると桁数までゼロ埋めされます (%05d)

コード

```
int Health = 50; // 体力の現在値
int MaxHealth = 100; // 体力の最大値（3桁あるね）

printf( “HP %03d / %03d”, Health, MaxHealth);
```

出力

```
HP 050 / 100
```

桁数の前にあえてゼロを付けるとゼロ埋めできる

%<桁数><型:d¦f¦lf> とすると桁数まで半角スペースで埋めます (%5d)
%0<桁数><型:d¦f¦lf> とすると桁数までゼロ埋めされます (%05d)
ただの「%」を出すには %% を入れます

コード

```
#include <cmath> // log10, ceil 関数を使えるようにする
int Health = 50;
int MaxHealth = 2000;
char Buffer[50]; // 出力する文字列を一時的に入れておくための場所
// 濃い紫のとこ(桁数)を入れる
sprintf(Buffer, "HP %%0%1$dd / %%0%1$dd", (int)ceil( log10( MaxHealth )));
// 薄い紫のとこ(体力の値)を入れる
printf(Buffer, Health, MaxHealth);
```

出力

```
HP 0050 / 2000
```

ある正の整数について、
log10をとり整数になるよう切り上げるとその整数の桁数になる
桁数の部分はprintfを2重に使い、MaxHealth の桁数に応じてゼロの数が変わるようにしています (あえて色分けしています)

%<桁数><型:d¦f¦lf>とすると桁数まで半角スペースで埋めます (%5d)
%0<桁数><型:d¦f¦lf>とすると桁数までゼロ埋めされます (%05d)
ただの「%」を出すには %% を入れます

コード

```
#include <cmath> // log10, ceil 関数を使えるようにする
int Health = 50;
int MaxHealth = 2000;
char Buffer[50]; // 出力する文字列を一時的に入れておくための場所
// 濃い紫のとこ(桁数)を入れる
sprintf(Buffer, "HP %%0%1$dd / %%0%1$dd", (int)ceil( log10( MaxHealth ) ));
// 薄い紫のとこ(体力の値)を入れる
printf(Buffer, Health, MaxHealth);
```

矢印の sprintf 関数の引数は…

char[]	引数1	Buffer	(printfでコンソールに出てくる結果を、かわりにこの変数に入れる)
char[]	引数2	"HP %%0%1$dd / %%0%1$dd"	
int	引数3	(int)ceil(log10(MaxHealth))	

> %<桁数><型:d¦f¦lf>とすると桁数まで半角スペースで埋めます (%5d)
> %0<桁数><型:d¦f¦lf>とすると桁数までゼロ埋めされます (%05d)
> ただの「%」を出すには %% を入れます

```
// 濃い紫のとこ(桁数)を入れる
sprintf(Buffer, "HP %%0%1$dd / %%0%1$dd", (int)ceil( log10( MaxHealth )));
```

→ 矢印の sprintf 関数を実行すると

char[]	引数1	Buffer	(printfでコンソールに出てくる結果を、かわりにこの変数に入れる)
char[]	引数2	"HP %%0%1$dd / %%0%1$dd"	
int	引数3	(int)ceil(log10(MaxHealth))	$= \mathrm{ceil}(\ log_{10} 2000\ ) = \mathrm{ceil}(3.301) = 4$

ceil(x): x よりは大きい最小の整数
ここでは 3.301 より大きい最小の整数、4

%<桁数><型:d¦f¦lf>とすると桁数まで半角スペースで埋めます (%5d)
%0<桁数><型:d¦f¦lf>とすると桁数までゼロ埋めされます (%05d)
ただの「%」を出すには %% を入れます

```
// 濃い紫のとこ(桁数)を入れる
sprintf(Buffer, "HP %%0%1$dd / %%0%1$dd", (int)ceil( log10( MaxHealth )));
```

→ 矢印の sprintf 関数を実行すると

char[] 引数1 Buffer (printfでコンソールに出てくる結果を、かわりにこの変数に入れる)

char[] 引数2 "HP %%0%1$dd / %%0%1$dd"

- "HP →
- %% → ただの「%」
- 0 → ゼロ
- %1$d → →文字列の後から1番目の変数の値
- d → d

int 引数3 (int)ceil(log10(MaxHealth))

ceil(x): x よりは大きい最小の整数
ここでは 3.301 より大きい最小の整数、4

$= \mathrm{ceil}(\ log_{10} 2000\ ) = \mathrm{ceil}(3.301) = 4$

> %<桁数><型:d¦f¦lf>とすると桁数まで半角スペースで埋めます (%5d)
> %0<桁数><型:d¦f¦lf>とすると桁数までゼロ埋めされます (%05d)
> ただの「%」を出すには %% を入れます

```
// 濃い紫のとこ(桁数)を入れる
sprintf(Buffer, "HP %%0%1$dd / %%0%1$dd", (int)ceil( log10( MaxHealth )));
```

→ 矢印の sprintf 関数を実行すると

char[]	引数1	Buffer (printfでコンソールに出てくる結果を、かわりにこの変数に入れる)
char[]	引数2	"HP %04d / %%0%1$dd"
int	引数3	(int)ceil(log10(MaxHealth))

"HP %04d / %%0%1$dd"

- %04d → d
- ただの「%」
- ゼロ
- %1$d → →文字列の後から1番目の変数の値

ceil(x): x よりは大きい最小の整数
ここでは 3.301 より大きい最小の整数、4

%<桁数><型:d¦f¦lf>とすると桁数まで半角スペースで埋めます (%5d)
%0<桁数><型:d¦f¦lf>とすると桁数までゼロ埋めされます (%05d)
ただの「%」を出すには %% を入れます

```
// 濃い紫のとこ(桁数)を入れる
sprintf(Buffer, "HP %%0%1$dd / %%0%1$dd", (int)ceil( log10( MaxHealth )));
```

→ 矢印の sprintf 関数を実行すると

char[] 引数1 Buffer (printfでコンソールに出てくる結果を、かわりにこの変数に入れる)

char[] 引数2 "HP %04d / %04d" → これが Buffer に入る

int 引数3 (int)ceil(log10(MaxHealth))

ceil(x): x よりは大きい最小の整数
ここでは 3.301 より大きい最小の整数、4

> %<桁数><型:d¦f¦lf>とすると桁数まで半角スペースで埋めます (%5d)
> %0<桁数><型:d¦f¦lf>とすると桁数までゼロ埋めされます (%05d)
> ただの「%」を出すには %% を入れます

コード

```
#include <cmath> // log10, ceil 関数を使えるようにする
int Health = 50;
int MaxHealth = 2000;
char Buffer[50]; // 出力する文字列を一時的に入れておくための場所
// 濃い紫のとこ(桁数)を入れる
sprintf(Buffer, "HP %%0%1$dd / %%0%1$dd", (int)ceil( log10( MaxHealth )));
// 薄い紫のとこ(体力の値)を入れる
printf(Buffer, Health, MaxHealth);
```

矢印の printf 関数の引数は…

char[]	引数1	Buffer	= "HP %04d / %04d"
char[]	引数2	Health	
int	引数3	MaxHealth	

出力

```
HP 0050 / 2000
```

コード

```
#include <cmath> // log10, ceil 関数を使えるようにする
int Health = 50;
int MaxHealth = 2000;

// 濃い紫のとこ(桁数)を入れる
sprintf(Buffer, "HP %%0%1$dd / %%0%1$dd", (int)ceil( log10( MaxHealth )));
// 薄い紫のとこ(体力の値)を入れる
printf(Buffer, Health, MaxHealth);
```

sprintf が変数に結果を格納するだけなのでこういう書き方をしなければなりませんが
C++ 20 からは std::format が使えるので、中間変数 Buffer を使わなくても1行で書けたりします

ただし、フォーマット文字列も桁数指定の仕方もまったく異なるので、printfと混同しがちです
std::format を使ってコンソール出力をするときは、シンプルに print か std::cout を使いましょう

string 型を printf で使うには、string 型のメソッド c_str() を使う必要があります

コード

```
#include <cmath> // log10, ceil 関数を使えるようにする
int Health = 50;
int MaxHealth = 2000;
printf(
  std::format(
    std::format("HP {0:0{0:d}d}/ {1:0{0:d}d}" , Health, MaxHealth);
  ), (int)ceil( log10( MaxHealth ))
);
```

sprintf が変数に結果を格納するだけなのでこういう書き方をしなければなりませんが
C++ 20 からは std::format が使えるので、中間変数 Buffer を使わなくても1行で書けたりします

ただし、フォーマット文字列の作り方が異なるので、printfと混同しがちです
どちらか片方に統一すると読みやすいコードになります

string 型を printf で使うには、string 型のメソッド c_str() を使う必要があります