入門講習会 再帰

1. 再帰

関数の中で同じ関数を呼び出すことを再帰という。再帰によって書かれた関数のことを再帰関数と呼ぶ。

再帰関数が使われるタイミングは様々であるが，再帰が有効だと思われる場面は

A） 同じような処理を繰り返したい

B） 再帰構造をもったものに対して何かの処理をしたい

C） 大きな問題をいくつかの小さな問題に分割して考えたい

の 3 点である。

2. 再帰の例

再帰を本格的に使った例は入門としては難しすぎると思われるので，今回は非常に簡単な例を示す。

[総和を求める関数]

1 から N までの総和を返す関数を作ってみよう。

```
#include <stdio.h>

int sum1toN(int N) {
    if(N == 1) return 1;
    return sum1toN(N - 1) + N;
}

int main(void)
{
    printf("%d¥n", sum1toN(10));
    return 0;
}
```

プログラムの動きを見てみよう。

まず，sum1toN(5)が呼び出されると，N=5 として関数内で処理

```
return sum1toN(N - 1) + N;
```

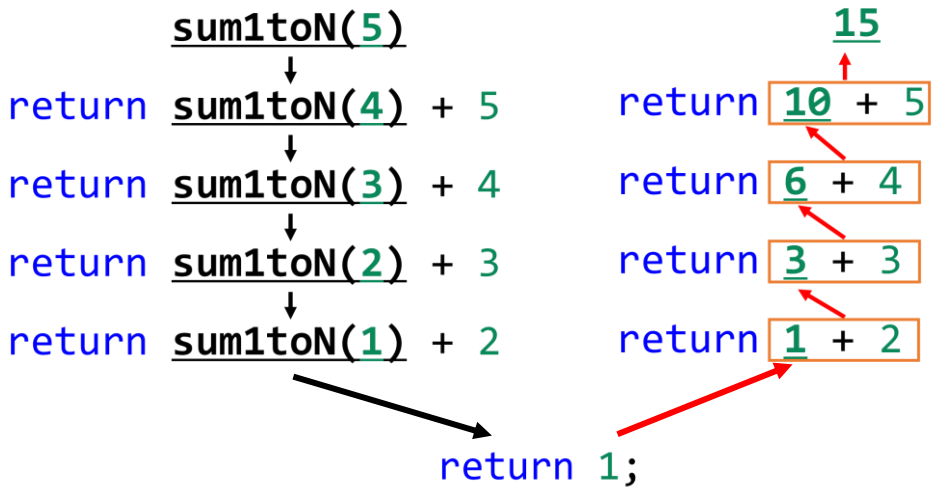
が行われる。N は 10 であるから，結局，

```
return sum1toN(4) + 5;
```

に読み替えられる。return で呼び出し元に戻りたいところだが，まだ sum1toN(4)がどんな値なのかわからないため，この関数を呼び出す。すると，N=4 として処理

```
  return sum1toN(N - 1) + N;
```
が行われる。N は 4 であるから，結局
```
  return sum1toN(3) + 4;
```
に読み替えられる。return で呼び出し元に戻りたいところだが，まだ sum1toN(3)がどんな値なのかわからないため，この関数を呼び出す。すると，N=3 として処理
```
  return sum1toN(N - 1) + N;
```
が行われる。N は 8 であるから，結局
```
  return sum1toN(2) + 3;
```
に読み替えられる。return で呼び出し元に戻りたいところだが，まだ sum1toN(2)がどんな値なのかわからないため，この関数を呼び出す。すると，N=2 として処理
```
  return sum1toN(N - 1) + N;
```
が行われる。N は 2 であるから，結局
```
  return sum1toN(1) + 2;
```
に読み替えられる。return で呼び出し元に戻りたいところだが，まだ sum1toN(1)がどんな値なのかわからないため，この関数を呼び出す。
N=1 となったとき，初めの if 文
```
  if(N == 1) return 1;
```
が真となり，ここで初めて return 文が処理される。sum1toN(1)の呼び出し元は，sum1toN(2)の return 部分だったから，
```
  return sum1toN(1) + 2;
```
は sum1toN(1)の返り値によって，
```
  return 1 + 2;
```
に読み替えられる。そして返り値 3 として呼び出し元 sum1toN(3)に返る。すると
```
  return sum1toN(2) + 3;
```
の部分は，sum1toN(2)の返り値によって，
```
  return 3 + 3;
```
に読み替えられる。そして返り値 6 として呼び出し元 sum1toN(4)に返る。すると
```
  return sum1toN(3) + 4;
```
の部分は，sum1toN(3)の返り値によって，
```
  return 6 + 4;
```
に読み替えられる。そして返り値 10 として呼び出し元 sum1toN(5)に返る。すると，
```
  return sum1toN(4) + 5;
```
の部分は，sum1toN(4)の返り値によって，
```
  return 10 + 5;
```
に読み替えられる。そして返り値 15 がようやく呼び出し元である main に返ってきて，printf で出力されることになる。

いままでの流れを図で表すと以下のようになる。黒矢印が関数呼び出しを，赤矢印が関数が返る処理を表している。



関数の中で関数を呼び出して，さらにその中で関数を呼び出して…という処理を行いながら，どんどん深くへと潜っていくイメージを持つと良い。そして最深部まで達すると，値を背負いながらもとの場所へと上がっていく。

さて，なぜ 1 から N までの総和を再帰で書くことができるのかを考えてみると，総和が再帰的な構造をしているからである。つまり，1 から N までの総和を S_N とすれば，

$$S_N = S_{N-1} + N \ (N = 2, 3, \cdots) \ , \ S_1 = 1$$

というように，S_N の中に S_{N-1} という同じ構造を含んでいるからである。一般に，漸化式で表されるものは再帰で書きやすい。

[階乗を計算する関数]

```c
#include <stdio.h>

int kaijo(int n) {
    if(n == 0 || n == 1) return 1;
    return kaijo(n - 1) * n;
}

int main(void)
{
    printf("%d¥n", kaijo(7));
    return 0;
}
```

実は前の例と大して変わらず, n の階乗を K_n とすれば，漸化式

$K_n = K_{n-1} \times n$ $(n = 2, 3, 4, \cdots)$, $K_0 = K_1 = 1$

が成り立つ。

末尾の n=0, 1 だけ例外的に if 文で処理させて，他の場合は関数を呼び出す。

[フィボナッチ数列]

int 型のサイズの都合上，このプログラムは第 40 項程度のフィボナッチ数まで求められる

```c
#include <stdio.h>

int fib(int n) {
    if(n == 0) return 0;
    if(n == 1) return 1;
    return fib(n - 1) + fib(n - 2);
}

int main(void)
{
    int n;
    scanf("%d", &n);
    printf("%d¥n", fib(n));
}
```

漸化式は, F_n を一般項として，

$F_n = F_{n-2} + F_{n-1}$ $(n = 2, 3, \cdots)$, $F_0 = F_1 = 1$

となっている。

さて，今回は簡単な例のため数列の一般項を再帰で書いたが，これ以外にも再帰が有用であるケースはたくさんある。むしろ，数列の一般項を求めるなら再帰よりもループで書いたほうがいいのだが(後述)，例のためにあえて再帰で書いた。

最後に，再帰について注意点を述べる。

慣れていないと再帰関数を書くのは難しいかもしれない。なぜなら，関数の中に自分と同じ処理をする関数があり，どんな動きをするのかが把握しづらいからだ。このようなときは，一旦，呼び出した再帰関数は「自分の思った通りに行ってくれる完璧な関数」であると思って書くと良い。その後に，その関数が適切に return してくれるように色々付け足す。

また，再帰の呼び出し回数を再帰の「深さ」と呼ぶことがある。再帰関数は深さ相当す

る情報を仮引数なりグローバル変数なりに保持していることが多い(もちろん深さの情報を持たなくても再帰が書けることはある)。
再帰の失敗パターンとして，適切に return 処理を書かなかったがために，永遠に再帰処理を行う無限ループに陥ってしまうことがある。必ず何かしらの場所で return 処理が行われるように関数を設計することを心がけよう。

3. 再帰とループの比較

前項の再帰の例を見て，「こんなものはループで書けるだろう」と思ったかもしれない。実際ループで書けて，そのほうが単純である(総和に関してはループすら使うまでもないのだが，説明の都合上以下はループで書かれている)。

[総和を求める関数(関数のみ)]

```
int sum1toN(int N) {
    int i;
    int ret = 0;
    for(i = 1; i <= N; i++) ret += i;
    return ret;
}
```

[階乗を計算する関数(関数のみ)]

```
int kaijo(int n) {
    int i;
    int ret = 1;
    for(i = 1; i <= N; i++) ret *= i;
    return ret;
}
```

[フィボナッチ数列(関数のみ)]

配列で前の項を保持しておいて，それを用いて次の項を計算する。これは動的計画法の一種であるが，詳しくは後期で述べる。

```
int fib(int n) {
    int i;
    int f[41];
    f[0] = 0;
    f[1] = 1;
    for(i = 2; i < 41; i++) f[i] = f[i - 2] + f[i - 1];
```

```
    return f[n];
}
```

ループは再帰に比べ，処理が早かったり，メモリが節約できたりすることが多い。もしも再帰なしで簡単に書けそうだったら，ループで書いたほうがよい。
しかし，ループでは書きにくかったり，また直感的には分かりにくかったりする場合がある。ループ文で書くか，再帰で書くかはその場の状況で判断する。