#04
マクロ, 列挙体, 再帰
2022 年度 / プログラミング及び実習 III

角川裕次

龍谷大学　先端理工学部

もくじ

今回 (#04) の内容

小テーマ: **マクロ**, **列挙体**, **再帰**

第 6 回： **関数呼び出しとスタック領域**

重要概念リスト

- 関数形式マクロ
- マクロ展開と関数の違い
- コンマ演算子
- バブルソートのアルゴリズム
- バブルソートの正しさ
- 列挙体
- 再帰呼び出し
- 関数呼出と復帰でのスタックの変化

今回の実習・課題 (manaba へ提出)

実習内容と課題内容は講義途中に提示します

(**作成したファイル類は** manaba **に提出**)

第 8-1 節　関数形式マクロ

□ 関数とマクロ p.228

int 型整数の 2 乗値を求める

関数で書いてみた

- 関数として呼び出しができる

```
int sqr_int(int x) {
  return x * x;
}
```

関数形式マクロで書いてみた

- マクロの定義：ソースコードの置き換え規則
- sqr_int(■) を ((■) * (■)) に置き換えてコンパイル

```
#define sqr_int(x)  ((x) * (y))
```

例: ソースコード中の `printf("n*n=%d\n", sqr_int(v));`
→ `printf("n*n=%d\n", ((v) * (v)));` に置き換え後にコンパイル

2 種類のマクロ定義：オブジェクト形式と関数形式

オブジェクト形式マクロ

引数のない記号だけのマクロ定義の形式

例：`#define NULL 0`

関数形式マクロ

引数を持ったマクロ定義の形式

- **引数の置き換えが行われる**

例：`#define sum_of(x,y) ((x)+(y))`

□ 関数と関数形式マクロ p.230

関数とマクロの比較 (例: 2 つの値の和を求める)

関数で書いてみた

```
int sum_of(int x, int y) {
  return x + y;
}
```

関数形式マクロで書いてみた

```
#define sum_of(x, y)  ((x)+(y))
```

関数形式マクロの定義と使用

マクロ定義の例

```
#define sum_of(x, y)  ((x)+(y))
```

マクロ使用のコードの例

```
printf("%d %d\n", sum_of(a, b), sum_of(c, 100));
```

... 以下はコンパイル作業の裏側...

このコードは
C プリプロセッサにより以下の通りマクロ展開される

- **引数の置き換えが行われる**

```
printf("%d %d\n", ((a)+(b)), ((c)+(100)));
```

そしてこれを C コンパイラ本体が機械語へとコンパイル

メモリ上ではどうなっている? 機械語プログラムの構成

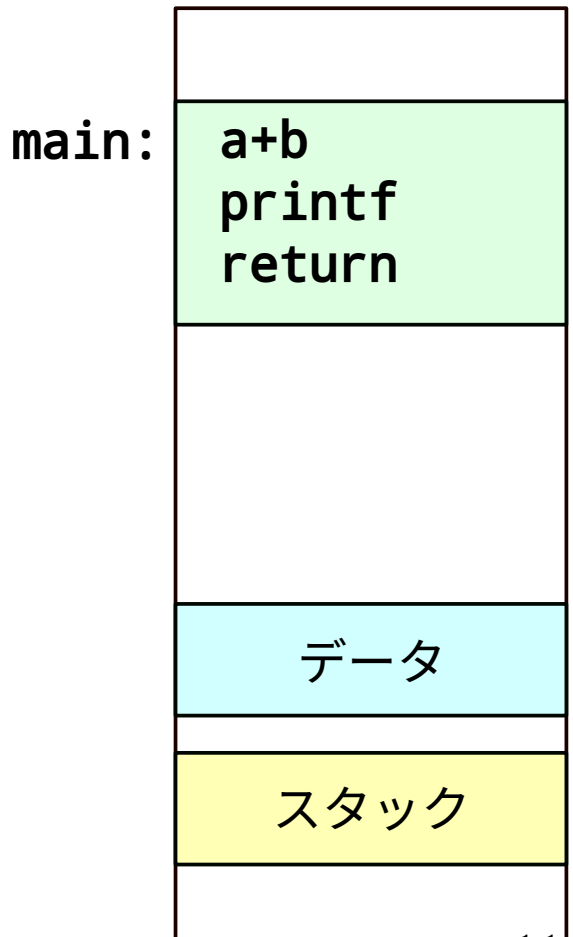
マクロ定義版

ソースコード

```
#define sum_of(x,y) ((x)+(y))
int main(void) {
  int a = 10; b = 92;
  printf("%d\n",
    sum_of(a, b));
  return 0;
}
```

以下の通りにマクロ展開されてコンパイル

```
int main(void) {
  int a = 10; b = 92;
  printf("%d\n",
    ((a)+(b)));
  return 0;
}
```



メモリ上ではどうなっている? 機械語プログラムの構成

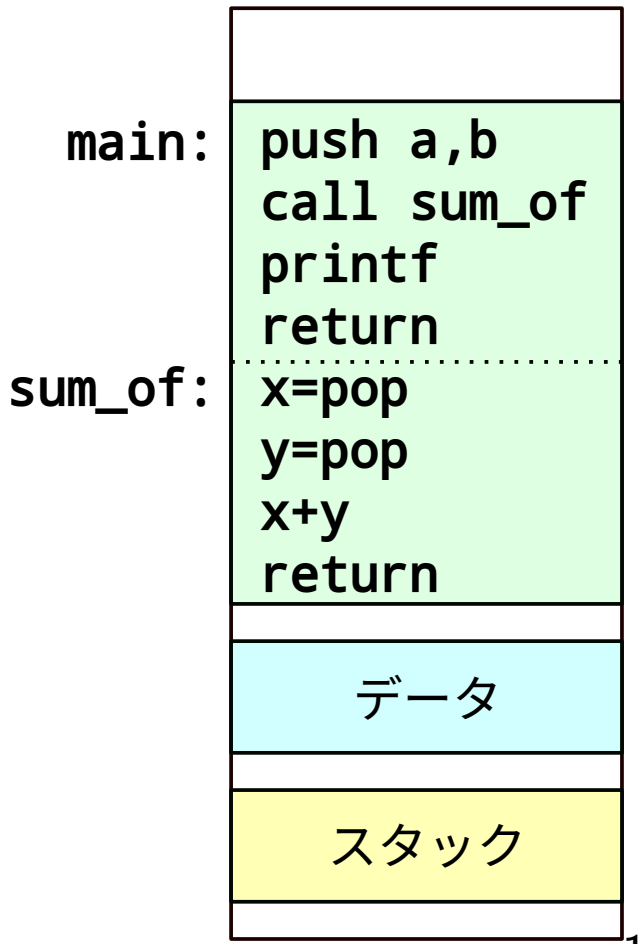
関数版

ソースコード

```
int main(void) {
  int a = 10; b = 92;
  printf("%d\n",
    sum_of(a, b));
  return 0;
}

int sum_of(int x, int y) {
  return x+y;
}
```

関数 sum_of **は機械語コードへと**
コンパイルされる



関数とマクロ：比較

★重要★

関数：関数に対応する機械語コードへとコンパイル

- C コンパイラ：関数の内容を機械語コードにコンパイル
- (実行時) 関数呼出：
  1. 復帰アドレスをスタックへプッシュ
  2. 関数が配置されている機械語のメモリアドレスへジャンプ
- (実行時) 関数を実行
- (実行時) 関数復帰：
  1. スタックより復帰アドレスをポップ
  2. 復帰アドレスへジャンプ

マクロ：マクロ利用箇所をマクロ定義に置き換えた後にコンパイル

- C プリプロセッサ：マクロの使用箇所をマクロ展開
- C コンパイラ：該当箇所を機械語コードにコンパイル
- (実行時) マクロ展開された機械語コードを実行
  (関数呼出/復帰に相当する動作はない)

マクロの落とし穴：展開後の形 (失敗例 1)

マクロ定義　(※やっちゃ駄目な定義)

```
#define sum_of(x,y)  x + y
```

マクロ利用

```
printf("%d\n", sum_of(a, b) * sum_of(c, d));
```

- **期待の動作**：$(a+b)(c+d)$ **の結果の表示**

実際には以下のものに展開される

```
printf("%d\n", a + b * c + d);
```

- **実際の動作**：$a+bc+d$ **の結果が表示**
- **「思うてんたと違う」**

マクロの落とし穴：展開後の形 (失敗例 2)

マクロ定義　(※やっちゃ駄目な定義)

```
#define sqr(x)  (x*x)
```

マクロ利用

```
printf("%d\n", sqr(a + b));
```

- **期待の動作**：$(a+b)^2$ **の表示**

実際には以下のものに展開される

```
printf("%d\n", a + b*a + b);
```

- **実際の動作**：$a+ba+b$ **の表示**
- **「思うてんたと違う」**

マクロの落とし穴：副作用 (失敗例 3)

マクロ定義　(※やっちゃ駄目な定義)

```
#define sqr(x)  ((x)*(x))
```

マクロ利用

```
printf("%d\n", sqr(a++));
```

- **期待の動作**：100 **が表示されて実行後は** a=11 **に**　(a=10 **の場合**)

実際には以下のように展開される

```
printf("%d\n", (a++)*(a++));
```

- **実際の動作**：110 **が表示されて実行後は** a=12 **に**
- **「思うてんたんと違う」**

マクロの落とし穴にはまらないために

- 大文字で書いて関数とビジュアルで区別つける
- 引数も大文字にする
- 定義内容の全体を括弧でくくる
- 引数それぞれを括弧でくくる

```
#define SQR(X)  ((X)*(X))
```

副作用を回避する定義... gcc **のみの非標準機能**

```
#define SQR(X) ({ typeof (X) x_ = (X); \
                  (x_ * x_) })
```

※長い定義はバックスラッシュ \ (または ¥) で改行できる

- 引数の値を最初に一度だけ計算
- その値を覚えておく
- その値を使ってマクロ本体の計算を行う

□ 引数のない関数形式マクロ p.231

引数なしの定義も可能

例

```
#define alert()  (putchar('\a'))
```

マクロ定義の書き方の注意

正しい例

```
#define sqr(x)    ((x)*(x))
```

駄目な例

```
#define sqr (x)   ((x)*(x))
```

理由：`sqr` と `(x)` **の間にスペースがあるため**

- `sqr` **を** `(x)  ((x)*(x))` **と定義している**
- **これはオブジェクト形式マクロの定義**
- **ソースコード中の** `sqr(a)` **は**
  `(x)  ((x)*(x))(a)` **へとマクロ展開される**

□ 関数形式マクロとコンマ演算子 p.232

関数形式マクロを活用する技能を紹介

よくやる間違い — どこが駄目?

```
#define puts_alert(str)  { putchar('\a'); puts(str); }

int main(void)
{
  int n;
  printf("整数を入力せよ：");
  scanf("%d", &n);
  if (n)
    puts_alert("その数はゼロではありません。");
  else
    puts_alert("その数はゼロです。");
  return 0;
}
```

マクロ展開してみると分かる

マクロ定義

```
#define puts_alert(str)  { putchar('\a'); puts(str); }
```

注目すべき元のソースコード部分

```
puts_alert("その数はゼロではありません。");
```

マクロ展開後

```
{ putchar('\a'); puts("その数はゼロではありません。"); };
```

`};` **で構文エラー**

コンマ演算子を用いてマクロ定義を工夫

マクロ定義

- **構造**：(**式**, **式**, ..., **式**)

```
#define puts_alert(str)  ( putchar('\a'), puts(str) )
```

展開前

```
if (n)
  puts_alert("その数はゼロではありません。");
else
  puts_alert("その数はゼロです。");
```

展開後 — 期待通りの動作

```
if (n)
  ( putchar('\a'), puts("その数はゼロではありません。") );
else
  ( putchar('\a'), puts("その数はゼロです。") );
```

コンマ演算子と式文

コンマ演算子：式, 式, 式, ..., 式

- 構文：複数の式をカンマで並べた形
- 全体がひとつの式
- 全体の値：最後の式の値

括弧：(式)

- ひとつの式を構成
- 値：括弧の内側の式の値

式文: 式;

- 式にセミコロンを付けると文になる

例：(式, 式); の形の式文

```
( putchar('\a'), puts("その数はゼロではありません。") );
```

第 8-2 節　ソート

□ バブルソート p.234

ソート：**大きさの順に並び替えること**

実行例 (データを 5 つ入力しそのソート結果を表示)

```
5人の身長を入力せよ。
 1番：179
 2番：163
 3番：175
 4番：178
 5番：173
昇順にソートしました。
 1番：163
 2番：173
 3番：175
 4番：178
 5番：179
```

バブルソートの main 関数部分

バブルソートアルゴリズムを用いて配列 a (5 **要素**) **を昇順にソート**

- **ソート後の結果** : $a[0] \leq a[1] \leq a[2] \leq a[3] \leq a[4]$

List 8-5 (**部分**) : main **関数**

```
#define NUMBER   5    /* 人数 */

int main(void)
{
  int height[NUMBER]; /* NUMBER人の学生の身長 */
  printf("%d人の身長を入力せよ。\n", NUMBER);
  for (int i = 0; i < NUMBER; i++) {
    printf("%2d番：", i + 1);
    scanf("%d", &height[i]);
  }
  bsort(height, NUMBER); /* ソート */
  puts("昇順にソートしました。");
  for (int i = 0; i < NUMBER; i++)
    printf("%2d番：%d\n", i + 1, height[i]);
  return 0;
}
```

ソート部分 (バブルソートアルゴリズム使用)

List 8-5 (**部分**) : **ソート部分**

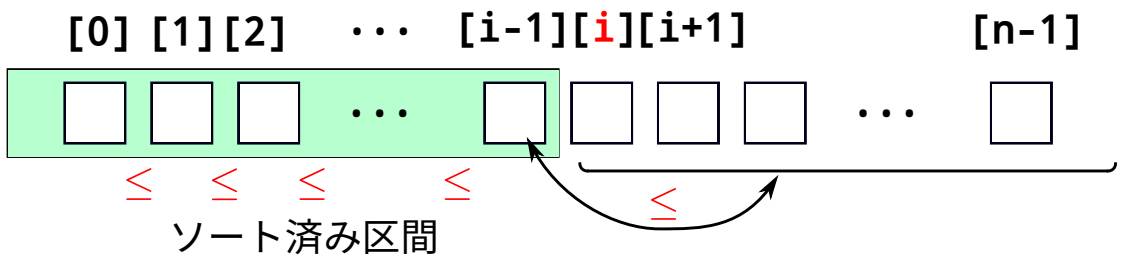
```
void bsort(int a[], int n)
{
  for (int i = 0; i < n - 1; i++) {
    for (int j = n - 1; j > i; j--) {
      if (a[j - 1] > a[j]) {
        int temp = a[j];
        a[j] = a[j - 1];
        a[j - 1] = temp;
      }
    }
  }
}
```

バブルソートの正しさ理解の基本アイディア

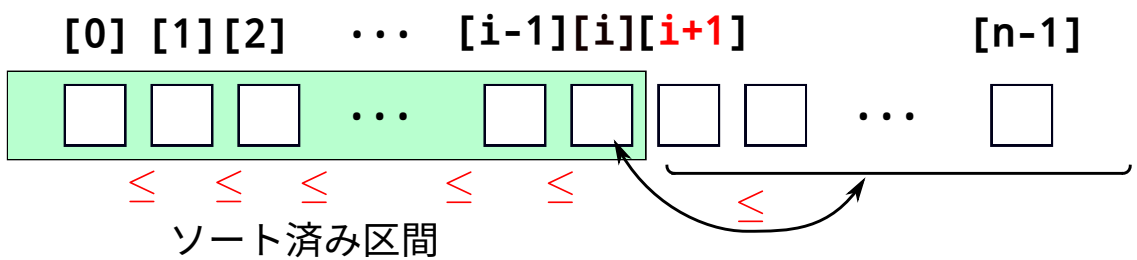
アルゴリズムの主要構造

```
for (int i = 0; i < n - 1; i++) {
  /* 命題 P(i) が成立 */
  動作;
  /* 命題 P(i+1) が成立 */
}
```

命題 $P(i)$

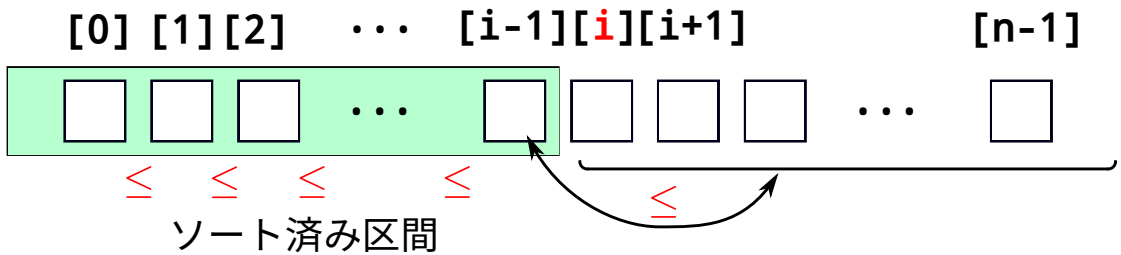


命題 $P(i+1)$

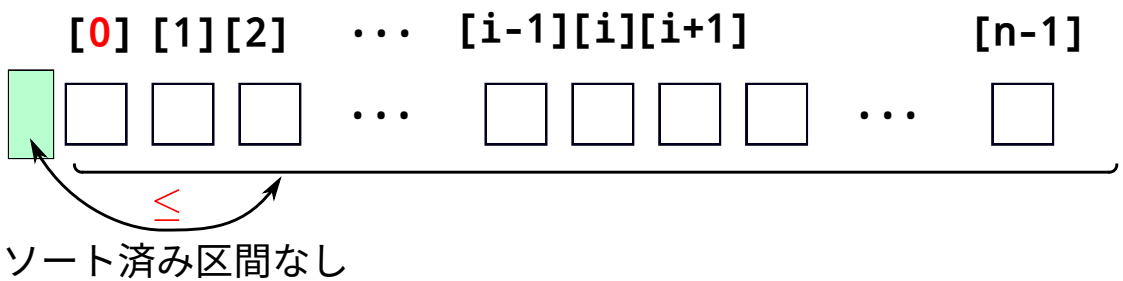


命題 $P(i)$ の定義

$a[0] \leq a[1] \leq \cdots \leq a[i-1]$ **が成立，かつ**
各 $k = i, i+1, \ldots, n-1$ **に対し** $a[i-1] \leq a[k]$ **が成立**



$P(0)$ **は自明に成立**

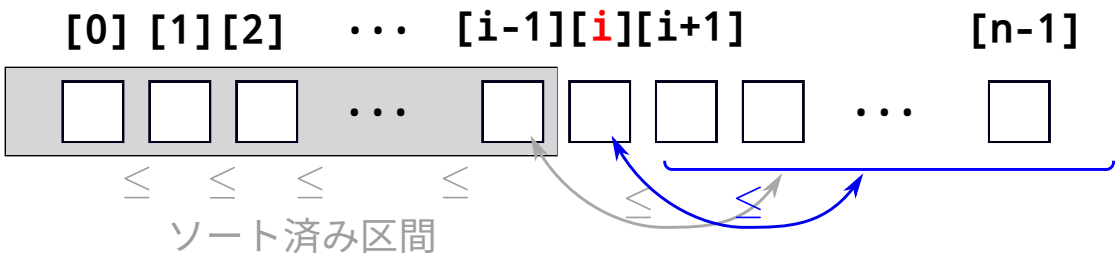


ループ内の動作の概要

```
for (int j = n - 1; j > i; j--) {
  if (a[j - 1] > a[j]) {
    a[j] と a[j-1] の 値 を 交 換;
  }
}
```

やっていること：$a[i] \leq a[k]$ $(\forall k = i+1, i+2, ..., n-1)$ **を成立させる**

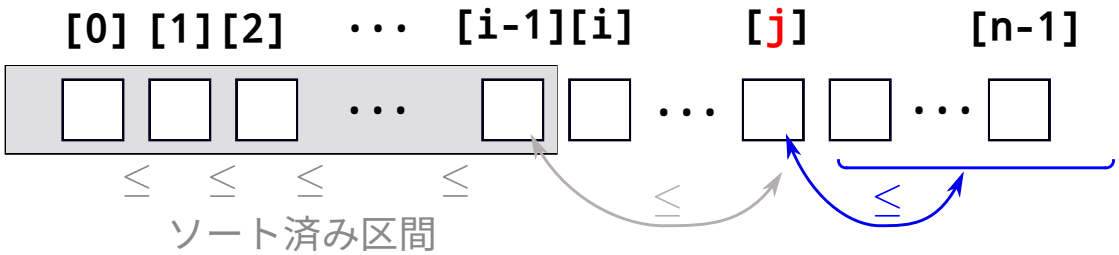
- **必要に応じて** $a[i], a[i+1], ..., a[n-1]$ **を並び替えて実現**



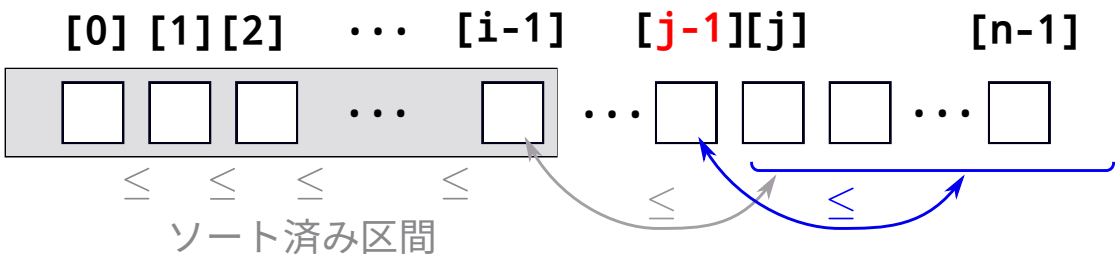
ループ内の動作の詳細 (1/2)

```
for (int j = n - 1; j > i; j--) {
  if (a[j - 1] > a[j]) {
    a[j] と a[j-1] の 値 を 交 換;
  }
}
```

一般的な j に対して成立すること (if **文の直前にて**)

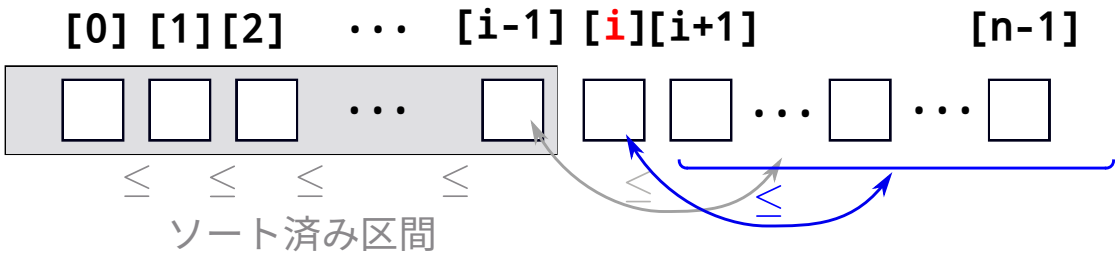


if **文の実行で成立する区間が** 1 **拡大**...

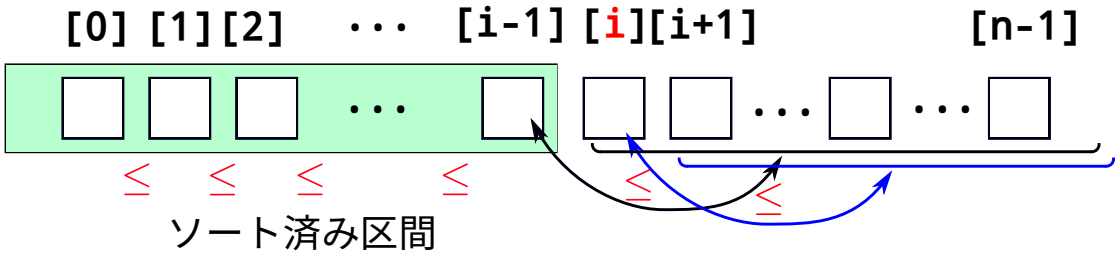


ループ内の動作の詳細 (2/2)

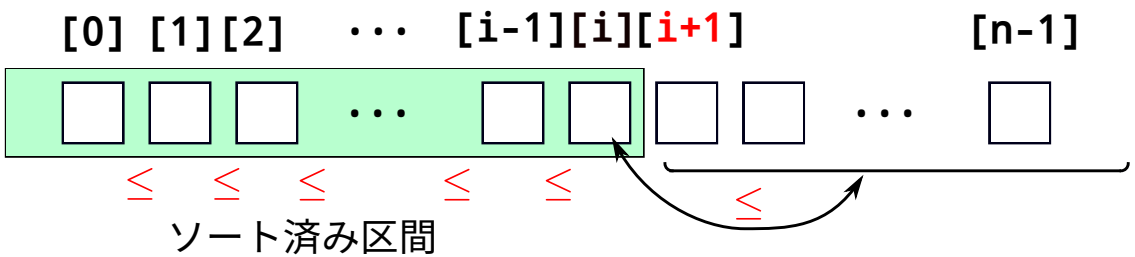
やがて $j = i + 1$ となる



これはすなわち



従って $P(i + 1)$ が if 文の実行後に成立 (ソート済み区間が 1 拡大)

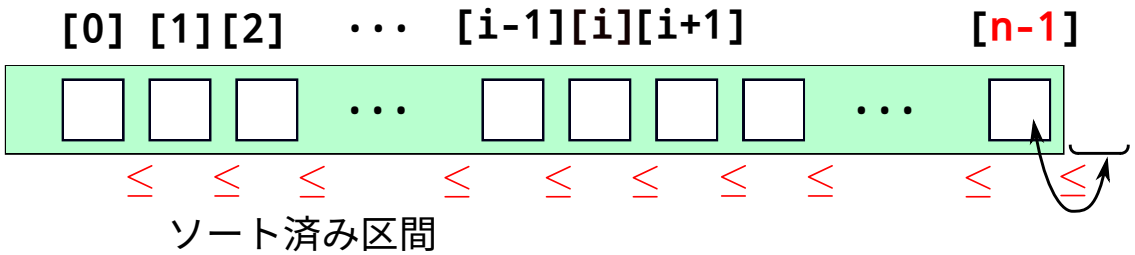


ループ終了時に成立すること

外側 for **ループの繰り返しを行う度にソート済み区間は** 1 **拡大**

```
for (i = 0; i < n - 1; i++) {
   /* 命題 P(i) が成立 */
   動作;
   /* 命題 P(i+1) が成立 */
}
```

外側 for **ループの中身を最後に実行するのは** $i = n - 2$ **のとき**
→ 命題 $P(n-1)$ **が成立**



$a[0] \leq a[1] \leq \cdots \leq a[n-2] \leq a[n-1]$ **が成立**
(**ソート完了**)

第 8-3 節　列挙体

□ 列挙体

列挙体 (enumeration)：**限られた整数値の集合を表すデータ型**

列挙型の定義の例：`enum animal`

```
enum animal { Dog, Cat, Monkey, Invalid }
```

- **新たな型** (enum animal) **の定義**
- `animal` **はタグ名と呼ばれる**
- **使える値** (**整数値**) **を** 4 **通りに限定**
- **値の指定に記号名の使用が可能**
- **使える記号名** (**列挙定数**) **を定義**：$\texttt{Dog} = 0, \texttt{Cat} = 1, \texttt{Monkey} = 2, \texttt{Invalid} = 3$

列挙型の変数宣言の例

```
enum animal sel;
```

列挙体を用いたプログラム例

List 8-6 (**部分**) : **キーボードから読み込む**

```
enum animal select(void)
{
  int tmp;
  do {
    printf("0… 犬  1… 猫  2… 猿  3… 終了：");
    scanf("%d", &tmp);
  } while (tmp < Dog || tmp > Invalid);
  return tmp;
}
```

- **動物えらびをする関数**
- scanf **で整数値を読む**
- enum animal **の範囲の値であればそれを返す**
- **範囲外なら再び読み直す**

列挙体を用いたプログラム例 (つづき)

List 8-6 (**部分**) : main **関数**

```
int main(void)
{
  enum animal selected;
  do {
    switch (selected = select()) {
    case Dog     : dog();     break;
    case Cat     : cat();     break;
    case Monkey  : monkey();  break;
    }
  } while (selected != Invalid);
  return 0;
}
```

- **関数** select **を呼び出して動物えらび**
- **動物毎に対応する関数を呼び出す**
- **ソースコードが読みやすくなり間違いしにくくなる**

列挙定数

列挙定数に明示的に値を設定可能

```
enum kyushu { Fukuoka, Saga = 5, Nagasaki };
```

- Fukuoka $= 0$ (**指定しなければ最初は** 0 **で始まる**)
- Saga $= 5$ (**指定するとその値になる**)
- Nagasaki $= 6$ (**指定しなければ前の値** $+1$ **になる**)

列挙体の便利さ

List 8-6 (**部分**; **改造**) : `enum animal` **を使わずに** `int` **で書いてみた**

```
int select(void)
{
  int tmp;
  do {
    printf("0… 犬  1… 猫  2… 猿  3… 終了：");
    scanf("%d", &tmp);
  } while (tmp < 0 || tmp > 3);
  return tmp;
}
```

可読性悪い

- **動物を追加するとこの関数も変更する必要あり**
- **おやくそく : 変更し忘れてバグ発生**

コンパイル時・実行時のチェックができない

- **扱いたい値は範囲が限定**
- `int` **型でコードを書くと値が範囲の内か否かを判定できない**

列挙体を使うと良い場合

取りうる値の種類が限定的

- 4 **種類** (0, 1, 2, 3)

それぞれの値に何らかの意味がある

- **犬**, **猫**, **猿**, Invalid

値は数値自体として意味がない (単なる区別のために使用)

- **犬を表す値が** 0 **であることに何も意味はない**

ソースコード上では記号名で値の表記ができる: 可読性が向上

□ 名前空間

列挙タグと変数名：同じ綴りの識別子を使って良い

```
int kyushu = 0;
enum kyushu { Fukuoka, Saga = 5, Nagasaki };
enum kyushu kp = kyusyu;
```

- **名前空間** (name space) **が異なるため区別される**
- **「**enum kyusyuu **型」の「変数** kp**」**

別の例

```
enum kyushu { Fukuoka, Saga = 5, Nagasaki };
enum kyushu kyusyu;
```

- **「**enum kyusyuu **型」の「変数** kyusyuu**」**

第 8-4 節　再帰的な関数

□ 関数と型

自然数の再帰的定義

- 1 は自然数
- 自然数の直後の整数も自然数

式の構文の再帰的定義 (の例)

- 自然数は式
- 式 + 式　も式
- 式 − 式　も式
- 式 × 式　も式
- 式 / 式　も式
- (式)　も式

□ 階乗値

定義：階乗 $n!$ $(n \geq 0)$

- $0! = 1$
- $n > 0$ **ならば** $n! = n \times (n-1)!$

例

$$
\begin{aligned}
5! &= 5 \times 4! \\
&= 5 \times 4 \times 3! \\
&= 5 \times 4 \times 3 \times 2! \\
&= 5 \times 4 \times 3 \times 2 \times 1! \\
&= 5 \times 4 \times 3 \times 2 \times 1 \times 0! \\
&= 5 \times 4 \times 3 \times 2 \times 1 \times 1 \\
&= 120
\end{aligned}
$$

階乗を計算する関数 factorial の呼出例

List 8-7 (**部分**)

```
int main(void)
{
  int num;
  printf("整数を入力せよ：");
  scanf("%d", &num);
  printf("%dの階乗は%dです。\n", num, factorial(num));
  return 0;
}
```

再帰呼び出しによる階乗を計算する関数

List 8-7 (**部分**)

```
int factorial(int n)
{
  if (n > 0)
    return n * factorial(n - 1);
  else
    return 1;
}
```

再帰呼び出しによる階乗を計算する関数

List 8-7 (**部分**)

```
int factorial(int n)
{
  if (n > 0)
    return n * factorial(n - 1);
  else
    return 1;
}
```

`factrial(3)` **の実行**

- `factrial(3)` **を呼出**
  - **仮引数** $\mathtt{n} = 3$
  - $\mathtt{n} \times \mathtt{factrial(2)}$ **を計算**

再帰呼び出しによる階乗を計算する関数

List 8-7 (**部分**)

```
int factorial(int n)
{
  if (n > 0)
    return n * factorial(n - 1);
  else
    return 1;
}
```

factrial(3) **の実行**

- factrial(3) **を呼出**
  - **仮引数** n = 3
  - n × factrial(2) **を計算**
- factrial(2) **を呼出**
  - **仮引数** n = 2

Q : n **は** 3 **から** 2 **に上書きされるの**?
A : **されません**

再帰呼び出しが正しく動作する理由

★重要★

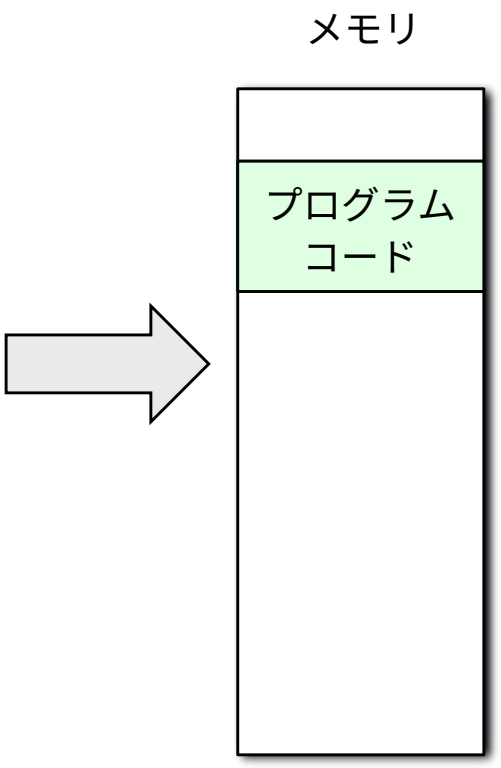
仮引数&自動変数用のメモリ場所：呼出毎に異なる場所を使用するため

実行開始前のメモリの様子

```
int main(void){
  int num = ...;
  printf("f(%d)=%d\n",
         num,
         fact(num));
  return 0;
}

int fact( int n ) {
  if (n > 0)
    return n * fact(n-1);
  else
    return 1;
  }
}
```

ソースコード



以降のスライド：実行過程でメモリが使用される様子を図示

実行過程： main に突入

自動変数 num をメモリに確保

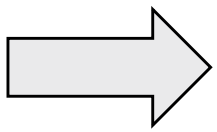
- **値 3 の場合で説明**

実行点

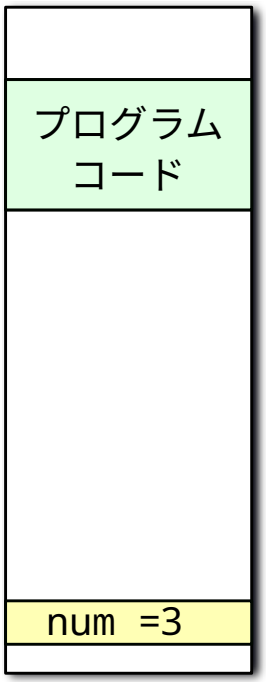
```
int main(void){
  int num = 3;
  printf("f(%d)=%d\n",
         num,
         fact(num));
  return 0;
}

int fact( int n ) {
  if (n > 0)
    return n * fact(n-1);
  else
    return 1;
  }
}
```

ソースコード



メモリ

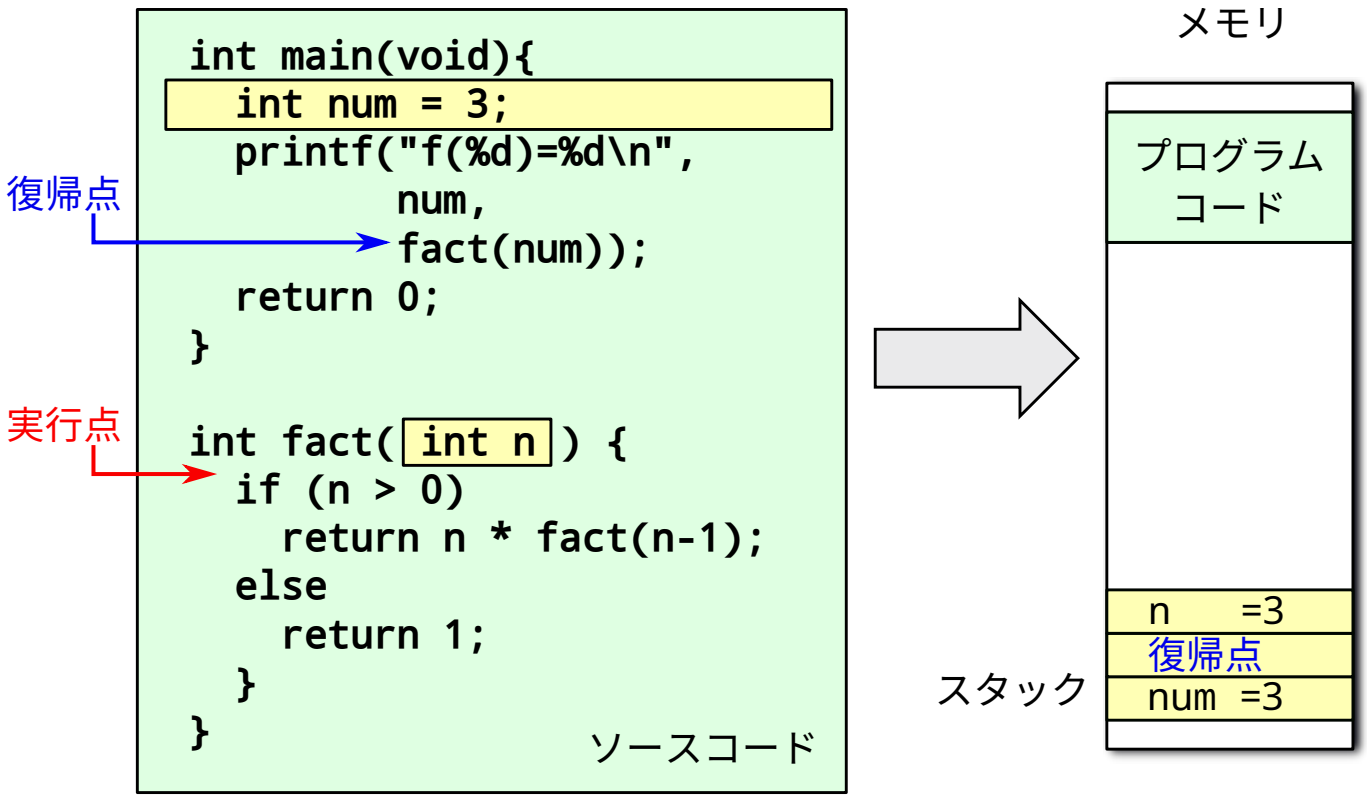


スタック

fact(3) **をこれから呼び出す**

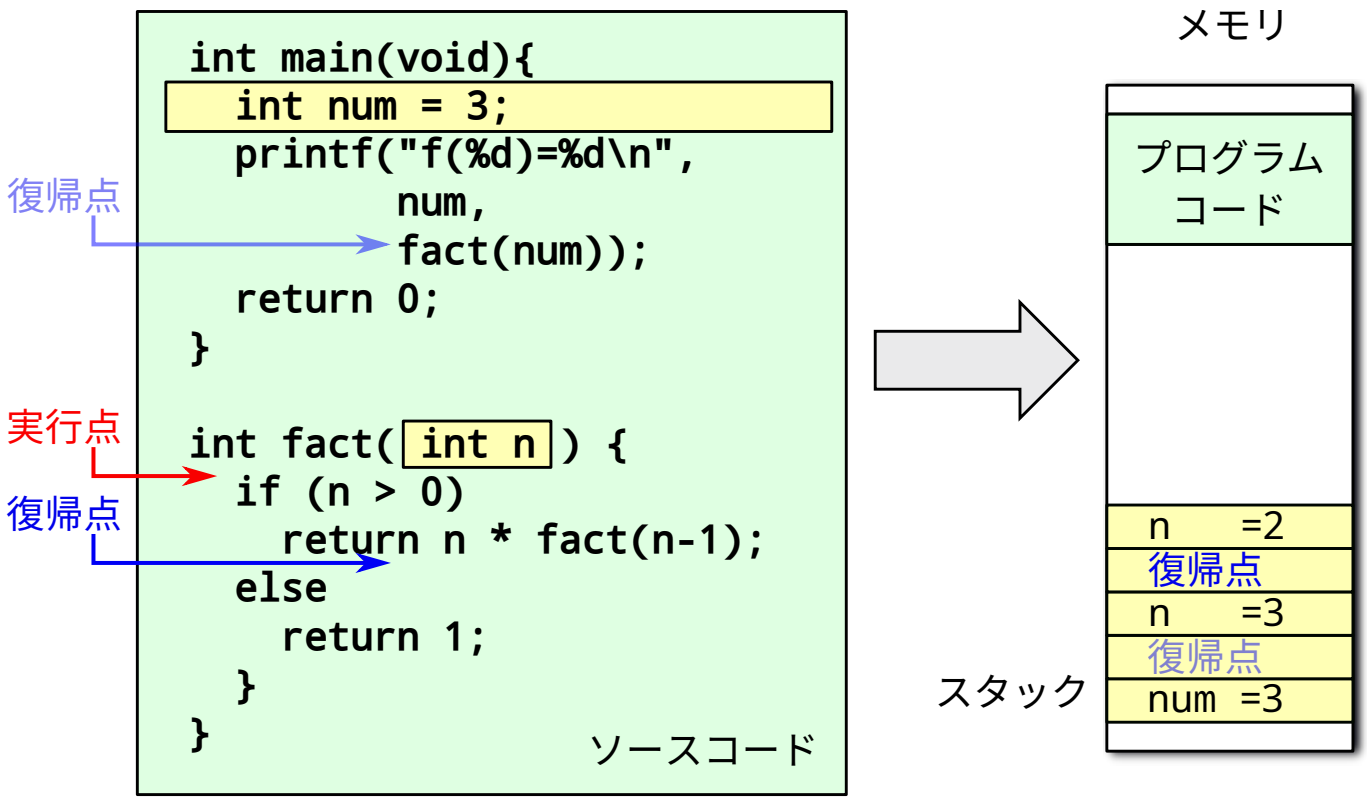
実行過程：fact(3) の呼出時

関数呼び出しの復帰点をスタックに保存
仮引数 n (=3) をメモリに確保



実行過程：fact(2) の呼出時

関数呼び出しの復帰点をスタックに保存
仮引数 n (=2) をメモリに確保

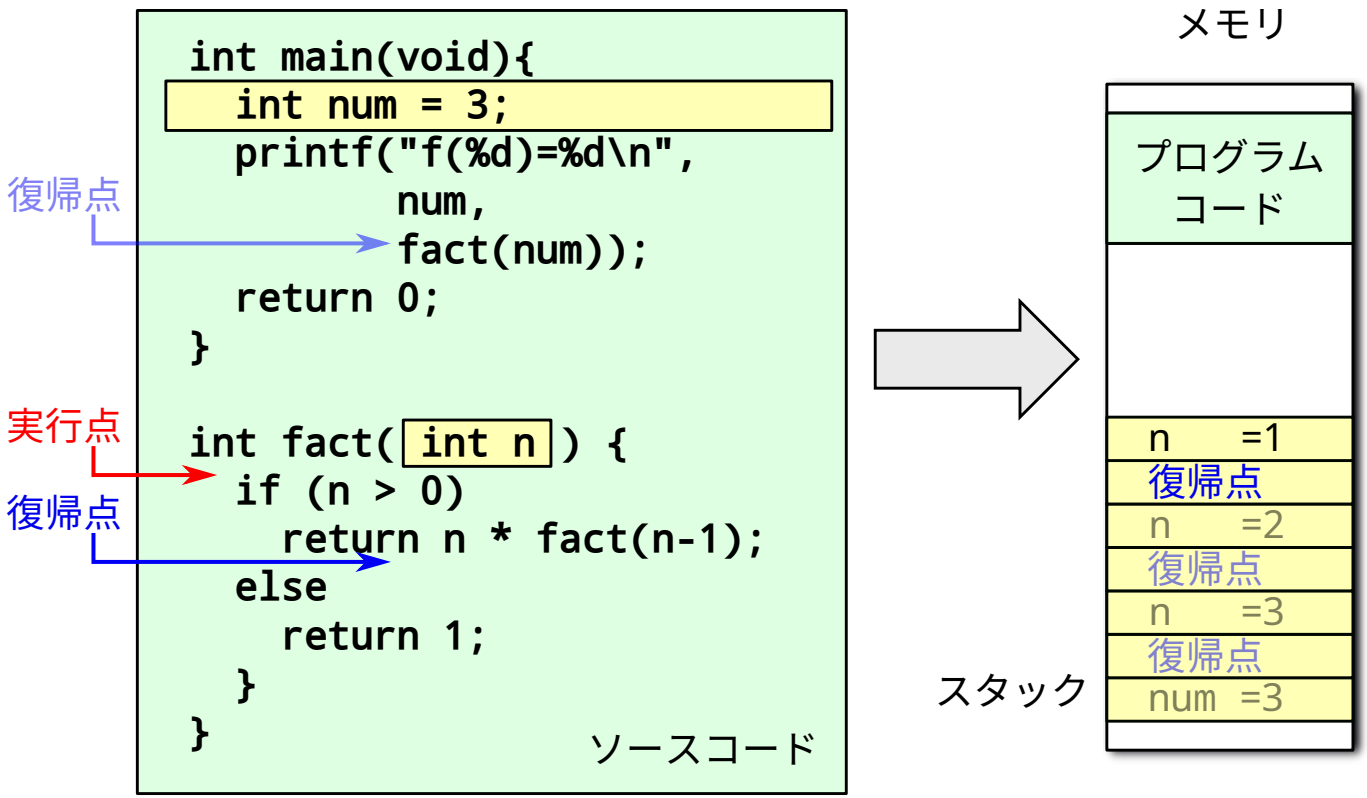


重要観察ポイント：呼び出し元の n の値が書き換わることはない

- **理由：再帰呼び出し毎に異なるメモリで n を記憶しているから**

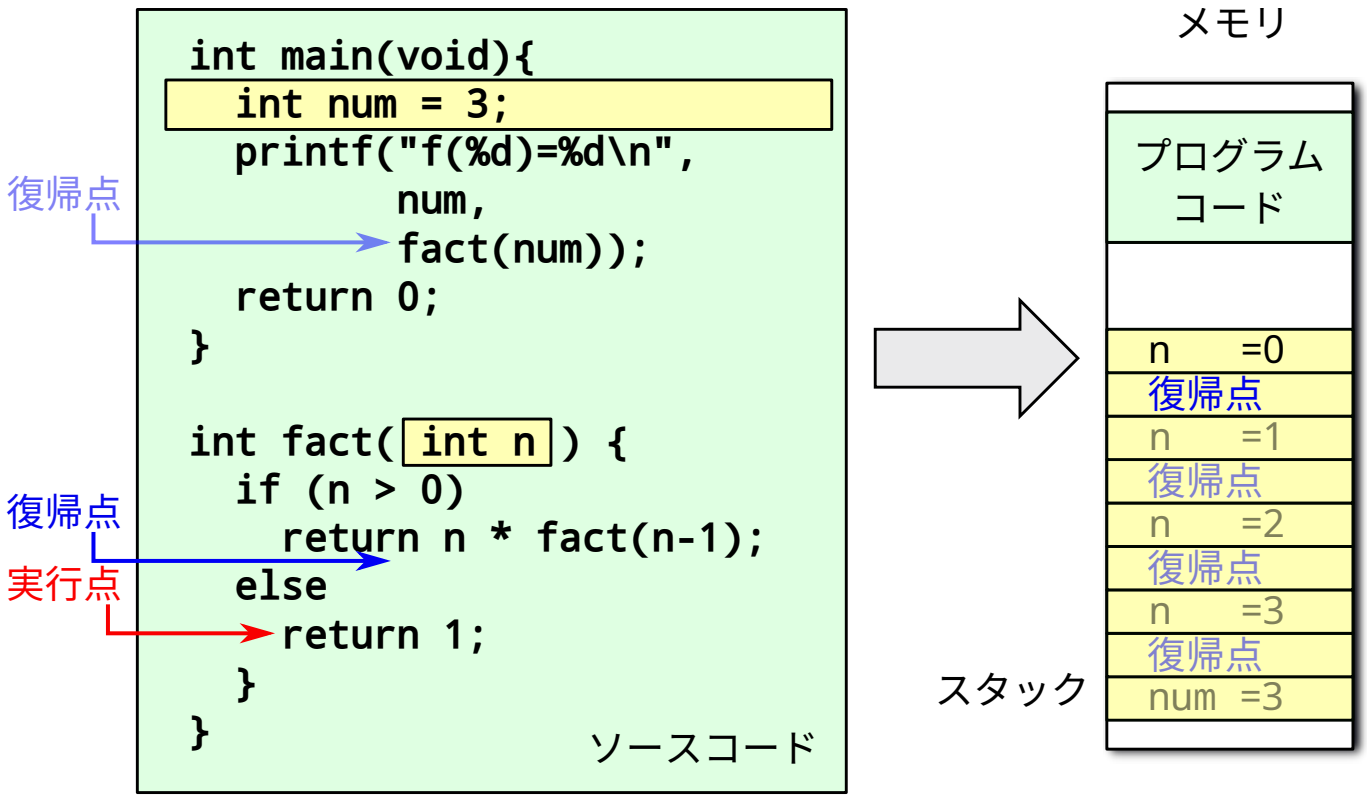
実行過程：fact(1) の呼出時

関数呼び出しの復帰点をスタックに保存
仮引数 n $(= 1)$ をメモリに確保



実行過程：fact(0) の呼出時

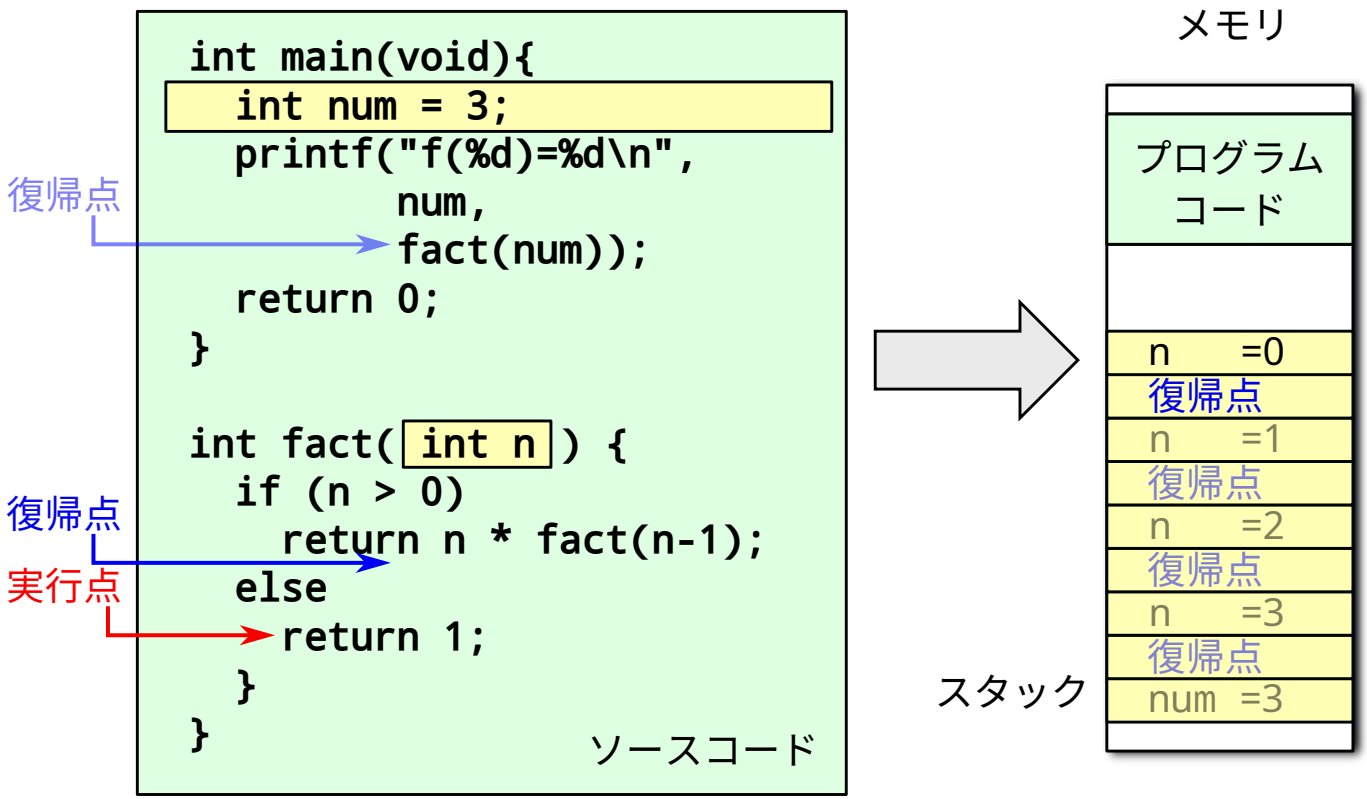
関数呼び出しの復帰点をスタックに保存
仮引数 n (= 0) をメモリに確保



n = 0 なのでこの後に return 1 が実行される

実行過程：fact(0) からの復帰時

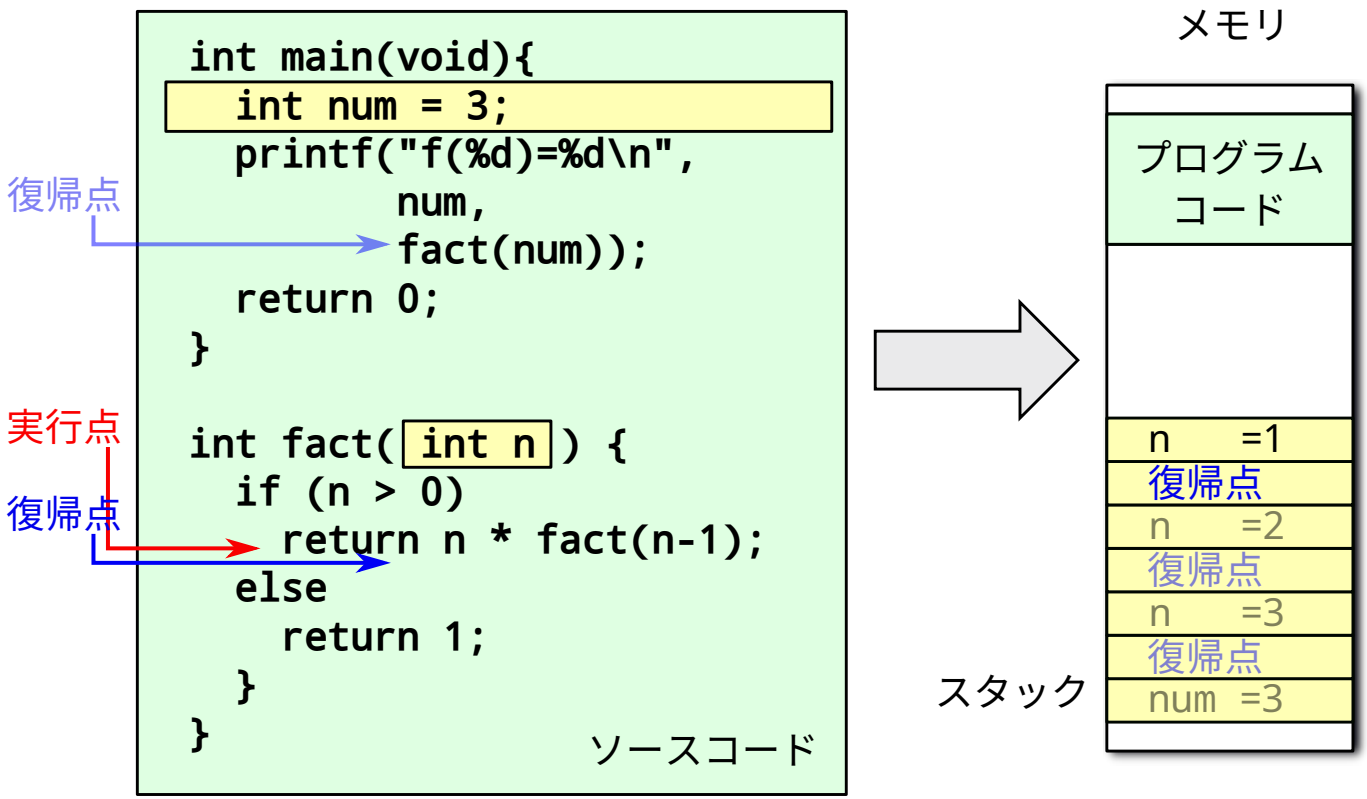
n = 0 なので関数値は 1
関数呼び出しの復帰点をスタックから取り出してジャンプ
スタックを巻き戻す



関数値 1 を返す

実行過程：fact(1) からの復帰時

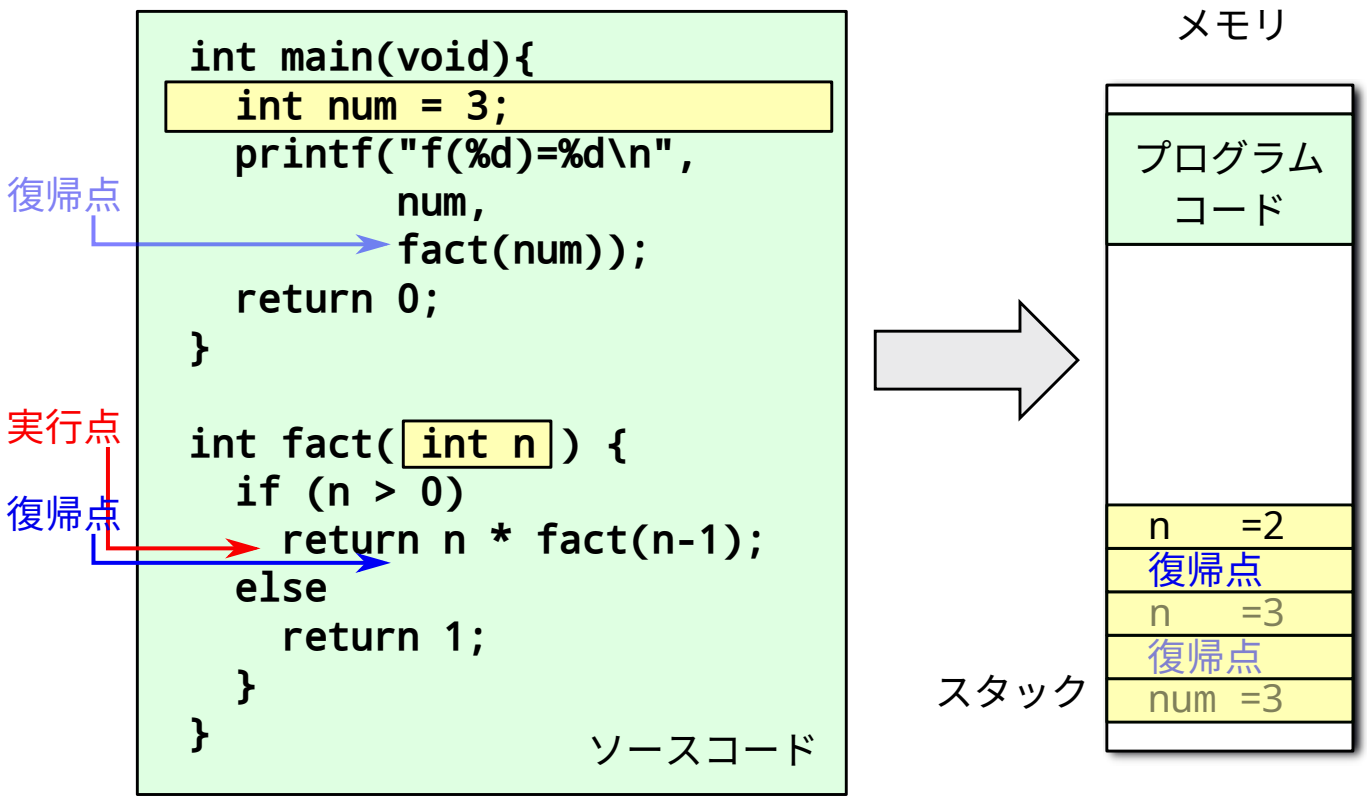
`n` $= 1$ **なので関数値は** `n * fact(0)` $= 1 \times 1 = 1$
関数呼び出しの復帰点をスタックから取り出してジャンプ
スタックを巻き戻す



関数値 1 **を返す**

実行過程：fact(2) からの復帰時

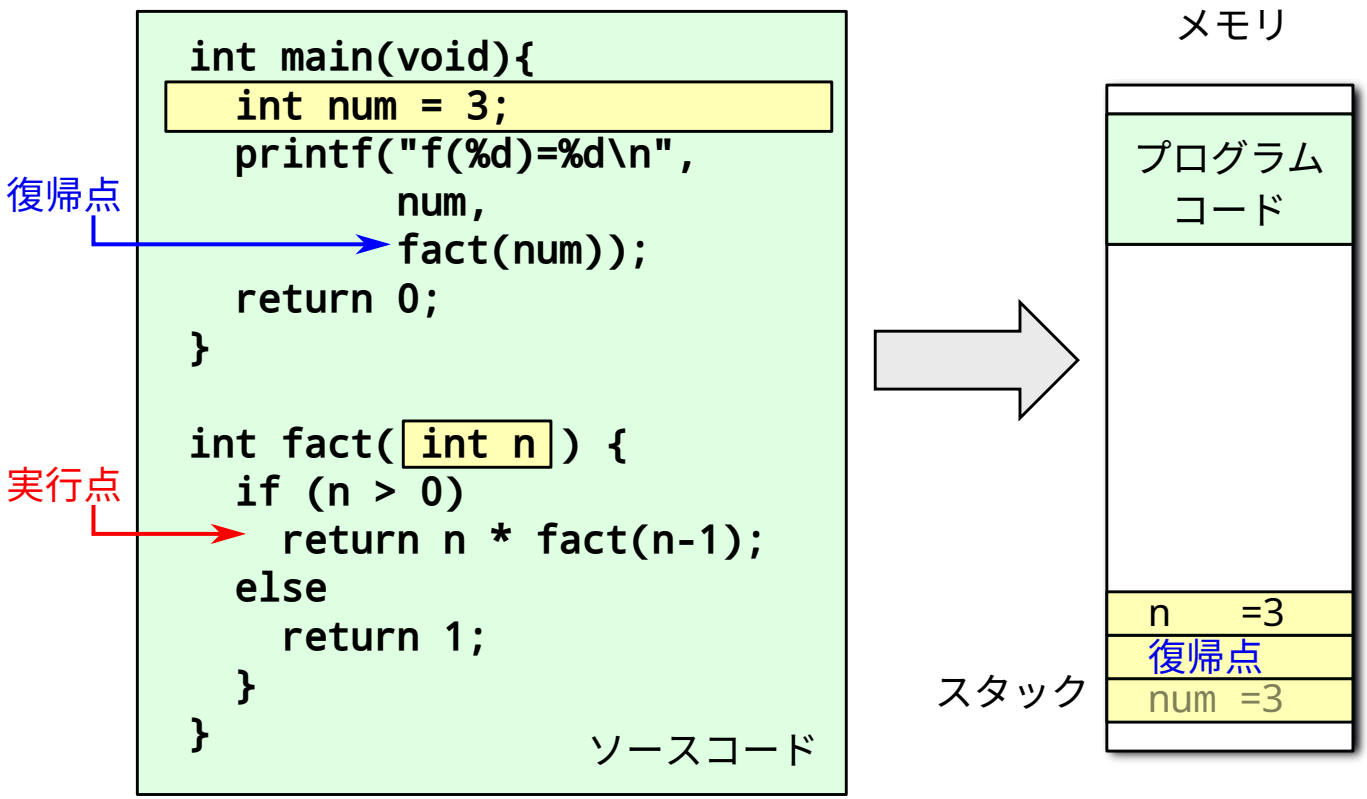
n = 2 **なので関数値は** n * fact(1) = 2 × 1 = 2
関数呼び出しの復帰点をスタックから取り出してジャンプ
スタックを巻き戻す



関数値 2 **を返す**

実行過程：fact(3) からの復帰時

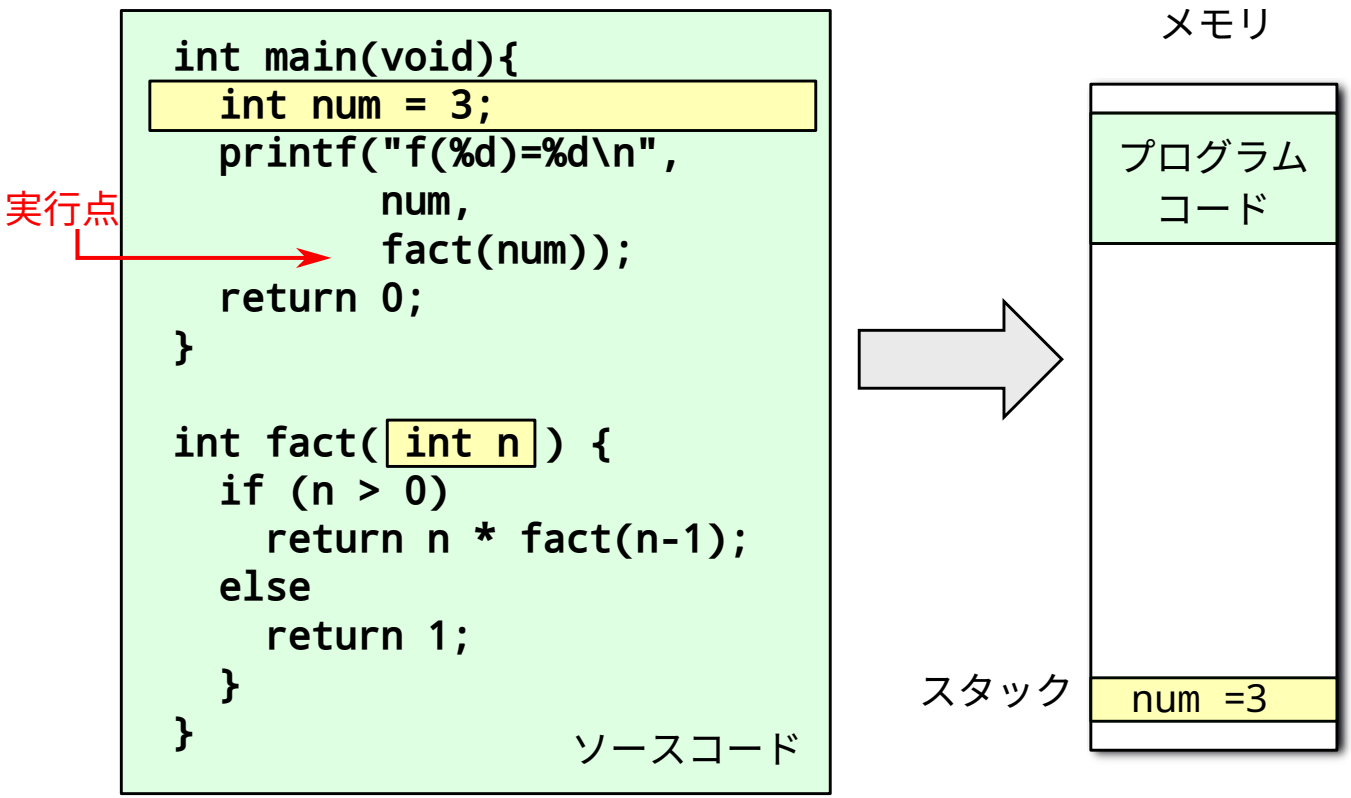
n = 3 **なので関数値は** n * fact(2) = 3 × 2 = 6
関数呼び出しの復帰点をスタックから取り出してジャンプ
スタックを巻き戻す



関数値 6 **を返す**

実行過程：fact(3) から復帰後

fact(3) = 6 を得て printf で表示



main 関数を終了
(main 関数の呼び出し元にリターン)

反復で階乗を計算する関数

実行速度の観点から普通はこう書く

```
int factorial(int n)
{
  int f = 1;
  int i = n;
  while (i > 0) {
    f = f * i;
    i = i - 1;
  }
  return f;
}
```

$1 \times n \times (n-1) \times (n-2) \times \cdots \times 2 \times 1$ **を計算**

おわり

番外編の課題 1

任意に与えられる正整数 n に対し、関数 $e(n)$ を以下の漸化式で (再帰的に) 定める。

$$\begin{aligned} e(0) &= 1 \\ e(n+1) &= 1 - e(n),\ n \geq 0 \end{aligned}$$

任意に与えられる正整数 n に対して $e(n)$ を (漸化式そのままに従って) 再帰呼び出しで計算するプログラムを作成しなさい。

番外編の課題 2

フィボナッチ数列は以下の漸化式で (再帰的に) 定義される。

$$
\begin{aligned}
f(0) &= 0 \\
f(1) &= 1 \\
f(n+2) &= f(n) + f(n+1),\ n \geq 0
\end{aligned}
$$

任意に与えられる正整数 n に対してフィボナッチ数 $f(n)$ を (漸化式そのままに従って) 再帰呼び出しで計算するプログラムを作成しなさい。

n が大きいと実行時間がかかるのはなぜか。その理由を考察せよ。