C++プログラミングI

- 第11回：再帰呼び出し
- 担当：二瓶芙巳雄

再帰呼び出しとは

- 用語
  - **再帰呼び出し** 自分自身を直接または間接的に呼び出す
  - **再帰的関数** 再帰呼び出しを含む関数
- 再帰呼び出しを使う理由
  - 数学的な考え方と相性がよい
  - プログラムを簡潔に表せる

階乗の計算（ループ）

- 定義：
  - $n! = 1 \times 2 \times 3 \times \cdots \times (n-1) \times n$
  - ただし、$0! = 1$
- 1 から `n` までの整数を初期値1の変数 `x` に掛けている
- `n` が 0 の場合も問題ない

```
1  int fact1(int n) {
2    int x {1};
3    for (int i=1; i<=n; i++)
4      x *= i;
5    return x;
6  }
7
8  int main() {
9    std::cout << fact1( 5 ) << "\n";
10 }
```

```
% ./a.out
120
```

階乗の計算（ループ，逆順）

- 定義
  - $n! = n \times (n-1) \times (n-2) \times \cdots \times 2 \times 1$
  - ただし、$0! = 1$
- `n=0` を最初に扱う
- `--n > 0` に注意 (-1した結果を比較する)
- 定義との対応が分かりづらい

```
1   int fact2(int n) {
2     if (n == 0) return 1;
3     int x {n};
4     while (--n > 0)
5       x *= n;
6     return x;
7   }
8
9   int main() {
10    std::cout << fact2( 5 ) << "\n";
11  }
```

```
% ./a.out
120
```

階乗の計算（再帰呼び出し）

- 漸化式の利用

$$n! = \begin{cases} 1 & \text{for } n = 0 \\ n \times (n-1)! & \text{for } n \geq 1 \end{cases}$$

```
1  int factorial(int n) {
2    if (n == 0) return 1;
3    return n * factorial( n-1 );
4  }
5
6  int main() {
7    std::cout << factorial( 5 ) << "\n";
8  }
```

```
% ./a.out
120
```

`factorial(2)` 実行の様子

```
// 2の階乗を計算する場合
factorial(2)
  return 2 * factorial(1);
  factorial(1)
      return 1 * factorial(0);
      factorial(0)
        return 1; // n == 0なので，1を返却
```

- `if` で再帰呼び出しの繰り返しを止める
- $2! = 2 \times 1 \times 1$

呼び出しを網羅する

```cpp
int factorial(int n) {
  if (n == 0) return 1;
  return n * factorial( n-1 );
}

int main() {
  std::cout << factorial( 5 ) << "\n";
}
```

`n`	関数呼び出し	`return` される値
5	`factorial(5)`	`5 * factorial(4)`
4	`factorial(4)`	`4 * factorial(3)`
3	`factorial(3)`	`3 * factorial(2)`
2	`factorial(2)`	`2 * factorial(1)`
1	`factorial(1)`	`1 * factorial(0)`
0	`factorial(0)`	`1`

- `factorial(5)` を呼ぶと，`return 5 * factorial(4);` で，`factorial(4)` が呼ばれる
  - `factorial(5)` の結果は，この時点では確定しない
- `factorial(4)` を呼ぶと，`return 4 * factorial(3);` で，`factorial(3)` が呼ばれる
  - `factorial(4)` の結果は，この時点では確定しない
- `factorial(0)` を呼ぶと，`1` が返却される
  - `factorial(1)` の結果が `1 * 1` で確定する

実行の流れを追う

```
1  // int factorial(int n) { // 書き換え前
2  //   if (n == 0) return 1;
3  //   return n * factorial( n-1 );
4  // }
5
6  int factorial(int n) { // 書き換え後
7    std::cout << "factorial( n=" << n << " )\n";
8    if (n == 0) {
9      std::cout << "  end" << "\n";
10     return 1;
11   }
12
13   int val { factorial( n-1 ) };
14   std::cout << "n=" << n
15             << " val=" << val << "\n";
16   return n * val;
17 }
```

```
18 int main() {
19   std::cout << factorial( 5 ) << "\n";
20 }
```

```
% ./a.out
factorial( n=5 )
factorial( n=4 )
factorial( n=3 )
factorial( n=2 )
factorial( n=1 )
factorial( n=0 )
  end
n=1 val=1
n=2 val=1
n=3 val=2
n=4 val=6
n=5 val=24
120
```

main関数と実行結果

- 3つの関数を試してみる

```
1  int main() {
2    for (int i = 0; i < 14; i++)
3      std::cout << i <<":"
4                << fact1(i) <<" "
5                << fact2(i) <<" "
6                << factorial(i) <<"\n";
7    return 0;
8  }
```

- 急激に増える点に注意する
- $13!$は誤り
  - 32ビット`int`の最大値`2147483647`
  - $13!$の結果`6227020800`を保存できなかった

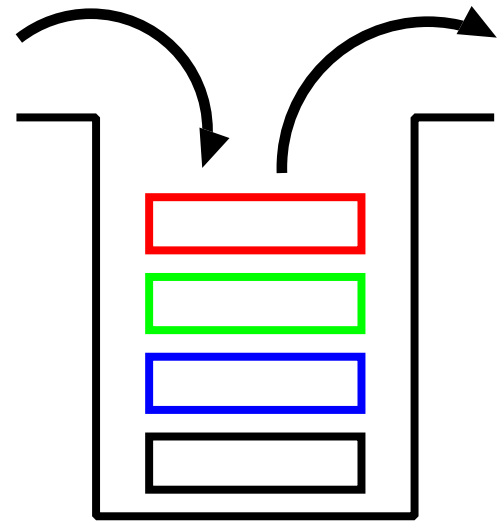
```
0:1 1 1
1:1 1 1
2:2 2 2
3:6 6 6
4:24 24 24
5:120 120 120
6:720 720 720
7:5040 5040 5040
8:40320 40320 40320
9:362880 362880 362880
10:3628800 3628800 3628800
11:39916800 39916800 39916800
12:479001600 479001600 479001600
13:1932053504 1932053504 1932053504
```

再帰呼び出しとループ

- 再帰呼び出しのプログラムはループで書ける
- 実行性能はループの方が再帰呼び出しより良い
- 再帰的関数で記述するとプログラムが簡潔
  - プログラムの開発期間
  - 維持コスト

再帰呼び出しと局所変数

- 関数呼び出しの度に仮引数や局所変数の場所が新たに割り当てられる
  - 自分自身を呼び出した後に仮引数や局所変数は上書きされない
  - スタックの構造が利用される
    - 後に入れたものを最初に出す
    - 積み上げた本や書類？



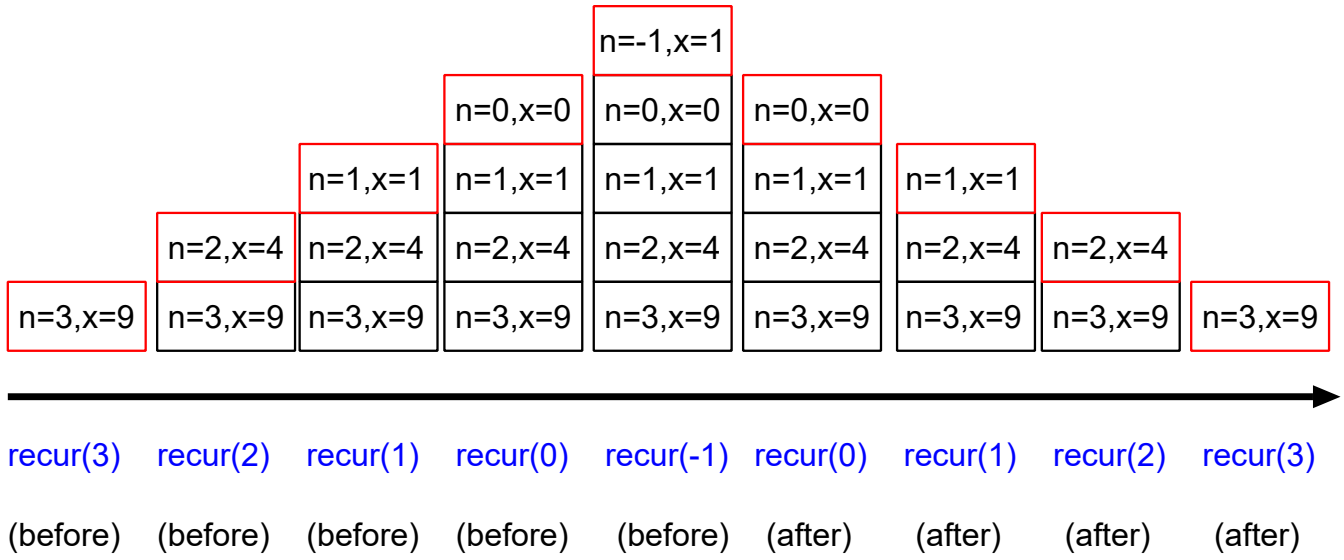
`recur(3)`の呼び出し

- 引数`n`と局所変数`x`は徐々に減ってまた増えている
- `n = -1`以外は`before/after`でペアになっている
  - ※変数`n`は，再帰によって上書きされていない
- `n = -1`は`if`文で再帰呼び出し前に`return`

```
1  void recur(int n) {
2    int x {n*n};
3    cout << n << ":" << x << "(before)\n";
4    if (n < 0) return;
5    recur( n-1 );
6    cout << n << ":" << x << "(after)\n";
7  }
8
9  int main() {
10   std::cout << recur( 3 ) << "\n";
11 }
```

```
% ./a.out
3:9(before)
2:4(before)
1:1(before)
0:0(before)
-1:1(before)
0:0(after)
1:1(after)
2:4(after)
3:9(after)
```

再帰の呼び出し図



- 一番上が実行中の関数
- 再帰呼び出しの度に段が積み上がる
- `return` の度に段を積み下ろす

入力と逆順の出力

- 入力した複数の文字列を入力とは逆の順序で出力
  - `"this is a pen"` → `"pen a is this"`
- `cin >> s` は入力成功で `true`
- 関数呼び出しごとに `s` が作られ、入力が保存される
- `EOF`（`CTRL+D`）で関数が戻り、出力が始まる
- ※再帰関数は戻り値がvoidでもOK，引数が無くてもOK

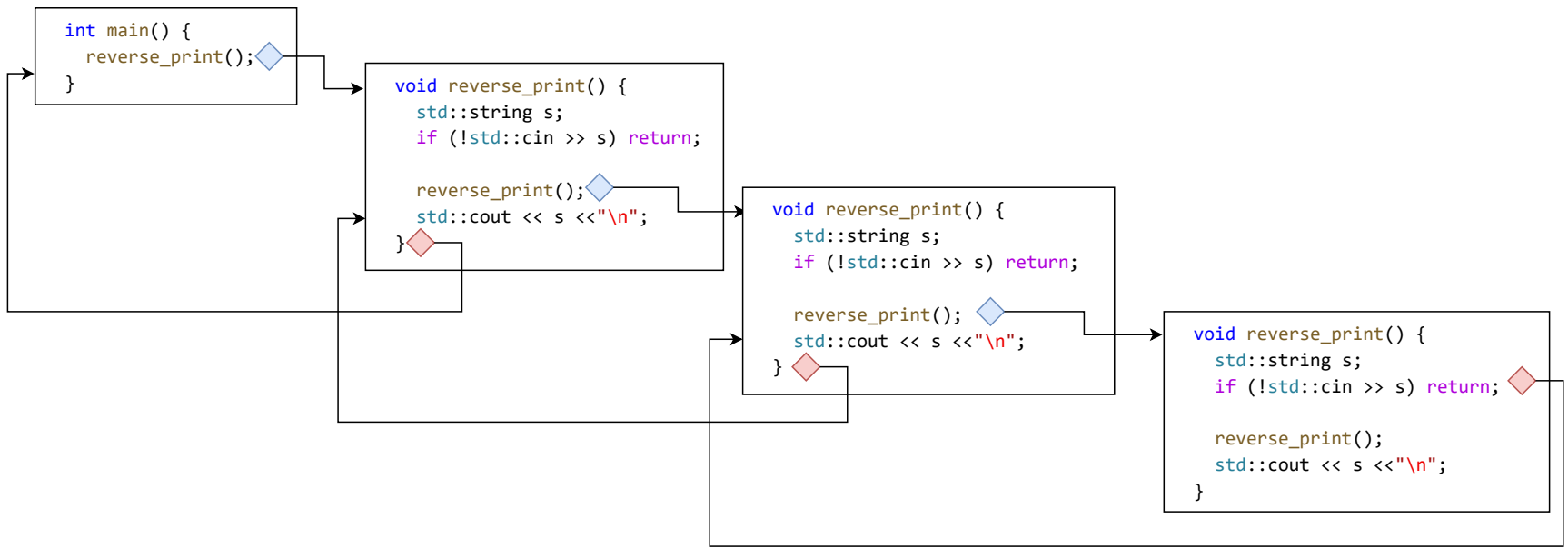
```
% ./a.out
this is a pen
(Ctrl + D)
pen a is this
```

```
1  void reverse_print() {
2    if (std::string s; std::cin >> s) {
3      reverse_print();
4      std::cout << s << " ";
5    }
6  }
7
8  int main() {
9    reverse_print();
10 }
```

```
1  // ↑を書き換え
2  void reverse_print() {
3    std::string s;
4    if (!std::cin >> s) return;
5
6    reverse_print();
7    std::cout << s << " ";
8  }
```

処理の流れ：補足

- `main()` から右下に流れ，`main()` に戻っていくイメージ
- `c++` → `programming` → EOF(Ctrl+D)の順で入力する場合は…



回文 (palindrome) の判定

- 回文：前から読んでも後ろから読んでも同じ文
- 定義を無理やり漸化式で表すと次のようになる

$$\text{n文字の回文} : \begin{cases} \text{空文字列} & (n = 0\text{の場合}) \\ \text{任意の1文字} & (n = 1\text{の場合}) \\ X\ (n-2\text{文字の回文})\ X & (n \geq 2\text{の場合}) \end{cases}$$

```cpp
bool is_palindrome(std::string s) {
  if (s.size() == 0) return true; // 文字数が偶数
  if (s.size() == 1) return true; // 文字数が奇数
  return (s.front() == s.back()) && is_palindrome( s.substr(1, s.size()-2) );

  // ↑の return 部分を書き換え↓
  // if( s.front() == s.back() ) return is_palindrome(s.substr(1, s.size()-2))
  // return false
}

int main() {
  std::cout << std::boolalpha << is_palindrome( "civic" ) << "\n";  // true
  std::cout << std::boolalpha << is_palindrome( "city" ) << "\n";   // false
}
```

回文の判定：空文字列の除外

- 定義の中で空文字列を使っている
- 元々の空文字列を除外する

```
% ./a.out
abccba is a palindrome.
abcdcba is a palindrome.
abcdecba is not a palindrome.
null string
```

```
1  bool is_palindrome(std::string s); // さっきのやつ
2  void check_palindrome(std::string s) {
3    if (s.empty()) {
4      std::cout << "null string\n";
5      return;
6    }
7
8    std::string x { is_palindrome(s) ? "" : "not " };
9    std::cout << s <<" is "<< x <<"a palindrome.\n";
10 }
11
12 int main() {
13   check_palindrome("abccba");
14   check_palindrome("abcdcba");
15   check_palindrome("abcdecba");
16   check_palindrome("");
17 }
```

簡単な再帰関数

- 終了条件は何か
- 終了条件に向かうように再帰呼び出ししているか
- ※各種エラー処理は省略しています

```
1   void bad_recursive( int i ) { // 終了条件がない！
2     std::cout << " bad: " << i;
3     bad_recursive( i+1 );
4   }
5
6   void print( int n ) { // nから0までの整数を表示
7     std::cout << n << " ";
8     if( n == 0 ) { // 終了条件
9       std::cout << "\n";
10      return;
11    }
12    print( n-1 ); // 終了条件に向かうように呼び出し
13  }
14
15  void print_to_ten( int n ) { // nから10までの整数を表示
16    std::cout << n << " ";
17    if( n == 10 ) { // 終了条件
18      std::cout << "\n";
19      return;
20    }
21    print_to_ten( n+1 ); // 終了条件に向かうように呼び出し
22  }
```

```
23  int sum( int n ) { // nから0までの整数の和を計算
24    if( n == 0 ) return 0; // 終了条件
25    return n + sum( n-1 ); // 終了条件に向かうように呼び出し
26  }
27
28  int sum_from_to( int n, int m ) { // nからmまでの整数の和を計算
29    if( n == m ) return n; // 終了条件
30    return n + sum_from_to( n+1, m ); // 終了条件に向かうように呼び出し
31  }
32
33  int count_even( int n ) { // nから0までの偶数をカウント
34    if( n == 0 ) return 1; // 終了条件
35    if( n%2 == 0 ) return 1 + count_even( n-1 );  // 終了条件に向かう
36    return 0 + count_even( n-1 );                 // ように呼び出し
37  }
38
39  int main() {
40    // bad_recursive( 0 ); // 無限ループ！Ctrl+Cで止める！
41    print( 7 );
42    print_to_ten( 3 );
43    std::cout << "sum() " << sum( 5 ) << "\n";
44    std::cout << "sum_from_to() " << sum_from_to( 2, 5 ) << "\n";
45    std::cout << "count_even() "<< count_even( 10 ) << "\n";
46  }
```

```
% ./a.out
7 6 5 4 3 2 1 0
3 4 5 6 7 8 9 10
sum() 15
sum_from_to() 14
count_even() 6
```

再帰を使うプログラムの応用：式の評価

- `"5+4*(5-2)-3"`のような，計算式を文字列としたものの，計算結果を得るプログラムを実装
  - ※計算をすることを**評価**という
- とても難しいので，ステップごとに実装
  1. 加減算の式（括弧`(` `)`を使わない）の評価
  2. 加減算の式の評価
  3. 四則演算の式の評価
  4. その他の実装
- これ以降のスライドでは，式の評価プログラムをひたすら説明
  - ソースコードはアップロード済み，動作させてみるとより理解が深まるでしょう

加減算の式（括弧`(` `)`を使わない）の評価

- 大域変数 `text` 中の式を解析して、その文字列が表す加減算の式の計算結果を返す
  - 一桁の整数のみを仮定．数字と演算子`+` `-`以外のスペース文字などは現れない．
- 変数
  - `text`：解析対象の文字列，`idx`：処理対象の文字の場所．

```
1   std::string text {"3+4-2+1"};
2   int idx {0};
```

- 加減算の式とは
  - 因数：一桁の数
  - 式　：因数に `+` 因数（or `-` 因数）が0回以上続く

```
3      +4      -2      +1
因数   +因数   -因数   +因数
```

この段階では，再帰的な定義ではない→再帰呼び出しにはならない

実装

- ソースコード全体
- ※教科書版とスライド版は若干差がある様子
  - ※長いプログラムを読めるように訓練しましょう

```
std::string data {"3+4-2+1"};
int idx {0};

void match(char x) {
  if (data[idx]==x) idx++;
  else std::cerr <<"error\n";
}

int getnum() {
  int n {data[idx] - '0'}; // '0'..'9'の文字を整数に変換
  idx++;
  return n;
}
```

```
int fact() { // 因数は数字
  char x { data[idx] };
  if (std::isdigit(x)) return getnum();
  std::cerr << "error\n";
  return 0;
}

int expr() { // 式は項 or 加減の 2 項演算
  int left { fact() };
  while ( true ) {
    char op = data[idx];
    if (op == '+') {
      match('+'); left += fact();
    }
    else if (op == '-') {
      match('-'); left -= fact();
    }
    else break;
  }
  return left;
}

int main() {
  std::cout << expr() << "\n";
  return 0;
}
```

match()

- 現在の文字`x`が`+` or `-` ならば`idx++`

```
1  void match(char x) {
2    if (data[idx]==x) idx++;
3    else std::cerr <<"error\n";
4  }
```

`getnum()`, `fact()`：数字を処理

- 次の文字が数字ならば数値に変換して返す
- （この部分を追加拡張していきます）
- それ以外はエラー

```
1  int getnum() {
2    int n {data[idx] - '0'}; // '0'..'9'の文字を整数に変換
3    idx++;
4    return n;
5  }
6
7  int fact() { // 因数は数字
8    char x { data[idx] };
9    if (std::isdigit(x)) return getnum();
10   std::cerr << "error\n";
11   return 0;
12 }
```

`expr()`：式の処理

- 式は数字と演算子(`+` `-`)が交互に出現
  - 「数字」の後、「±数字」が0回以上繰り返される
- 最初に`fact()`を呼び, 続く文字が演算子ならば再度`fact()`を呼んで二つの`fact()`の結果に対して演算

```
1  int expr() { // 式は項 or 加減の 2 項演算
2    int left { fact() };
3    while ( true ) {
4      char op = data[idx];
5      if (op == '+') { match('+'); left += fact(); }
6      else if (op == '-') { match('-'); left -= fact(); }
7      else break;
8    }
9    return left;
10 }
```

括弧付き加減算の式の定義

- `"4+3+(2+1)"`のような式を対象にするには？
  - `(2-1)`を`a`と置き換えると全体は`4+3+a`という形で、括弧の式`(2-1)`は因数
- 括弧付き加減算の式とは
  - 因数：一桁の数 **または** `(` **式** `)`
  - 式　：因数に `+` 因数（or `-` 因数）が0回以上続く

```
4       +3       +(2+1)
因数   +因数    +因数

( 2         +1        )
  因数      +因数
```

再帰的な定義 → 再帰呼び出しとなる

括弧付きの加減算の式

- 数字の場所に「`(式)`」を許す
- `fact()`の処理で先に括弧内の式を評価
- `expr()`が`fact()`を呼び出し、`fact()`が`expr()`を再帰的に呼び出す形に

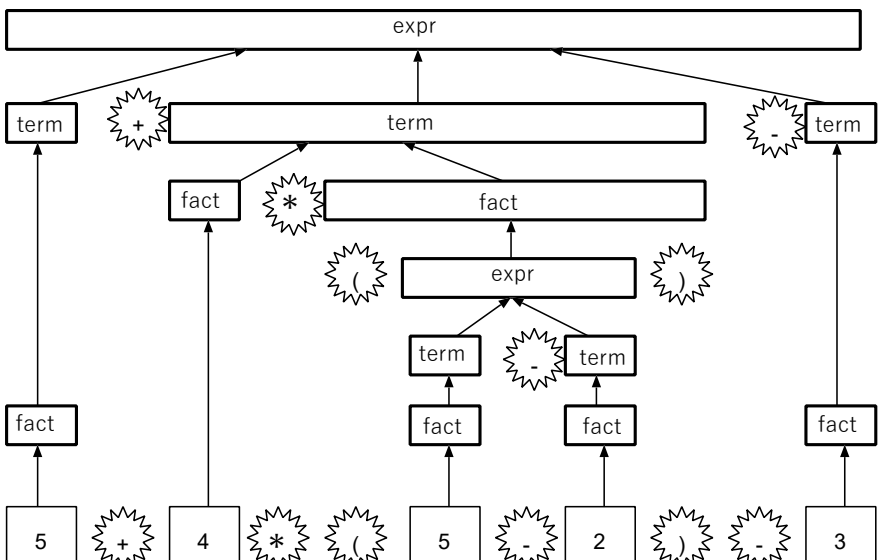
```
1  int fact() {
2    char x { data[idx] };
3    if (std::isdigit(x)) { return getnum(); }
4    else if (x == '(') {
5        match('(');
6        int n { expr() };   // 括弧内の部分式の処理
7        match(')');
8        return n;
9    }
10
11   std::cerr << "error\n";
12   return 0;
13 }
```

四則演算の式の評価

- `"5+4*(5-2)-9/3"` のような式を評価するには？
  - 乗除算の演算子(`*` `/`)は加減算の演算子(`+` `-`)よりも優先
  - 括弧内の式は、その外側のどの演算子よりも優先
- 用語
  - 因数(factor)：数字と括弧の式
  - 項 (term)：乗除算
  - 式 (expression)：加減算
- 定義
  - **fact**: 数字 または `( expr )`
  - **term**: `fact` に `*fact`（または `/fact`）が 0 回以上
  - **expr**: `term` に `+term`（または `-term`）が 0 回以上

式を定義に当てはめる

- `5+4*(5-2)-3` の構造
- 全体は「term+term-term」の形 → `5` + `4*(5-2)` - `3`
- 第2項は「fact *fact」の形 → `4` * `(5-2)`
- 第2項の後側のfact は「(expr)」という形、その式は「term-term」の形 → `5` - `2`



式と項の処理 `term()` / `expr()`

- `fact()` で数字と括弧付き式が処理される(前出)
- `term()` で `*` `/` の繰り返し,
  `expr()` で `+` `-` の繰り返しが処理

```
1  int term() {
2    int left { fact() };
3    while (char op = data[idx]){
4      if (op == '*') { match('*'); left *= fact(); }
5      else if (op == '/') { match('/'); left /= fact(); }
6      else break;
7    }
8    return left;
9  }
10
11 int expr() {
12   int left { term() };
13   while ( char op = data[idx] ) {
14     if (op == '+') { match('+'); left += term(); }
15     else if (op == '-') { match('-'); left -= term(); }
16     else break;
17   }
18   return left;
19 }
```

実装全体

- ソースコード全体

```
std::string data{"3+4*5-(6*7+8/2)"};
int idx{0};

int expr();  // 関数宣言，fact()から再帰的に呼ばれる
void match(char x) {
  if (data[idx] == x) idx++;
  else std::cerr << "error\n";
}
int getnum() {
  int n{data[idx] - '0'}; idx++;
  return n;
}
int fact() { // 因数は数字 or 括弧付きの式
  char x{data[idx]};
  if (std::isdigit(x)) { return getnum(); }
  else if (x == '(') {
    match('('); int n{expr()}; match(')');
    return n;
  }
  std::cerr << "error\n"; return 0;
}
```

```
int term() { // 項は因数 or 乗除の 2 項演算
  int left{fact()};

  while (char op = data[idx]) {
    if (op == '*') { match('*'); left *= fact(); }
    else if (op == '/') { match('/'); left /= fact(); }
    else break;
  }
  return left;
}

int expr() { // 式は項 or 加減の 2 項演算
  int left{term()};

  while (char op = data[idx]) {
    if (op == '+') { match('+'); left += term(); }
    else if (op == '-') { match('-'); left -= term(); }
    else break;
  }
  return left;
}

int main() {
  std::cout << expr() << "\n";
  return 0;
}
```

複数桁数字とホワイトスペースへの対応

- 複数桁数字への対応

```
int getnum() {
  int n {0};
  // 数字が続く限り10進数の値を計算する
  while (std::isdigit(data[idx])) {
    n *= 10;
    n += data[idx++] - '0';
  }
  return n;
}
```

- ホワイトスペースの除外

```
void skip_space()  {
  // ホワイトスペースを除外する
  while (std::isspace(data[idx]))
    idx++;
}
```

インタプリタに向けて

- 入力した文字列を式として解釈して、計算した結果を出力することがインタプリタの基本的な役割
- `input` 関数： `data` への入力を行う
- `eos` 関数： 入力した文字（式）をすべて解析したかどうかを確認（終端記号 `\0` まで解析が進んだかで判定）

```
1  int main(){
2    while ( input() ) {
3      std::cout << expr();
4      std::cout << ( eos()?"\n":" error\n" );
5    }
6    return 0;
7  }
```

完璧を目指すならば

- 0除算のチェック
- 単項演算の `+-`（`+3` や `-5` など）