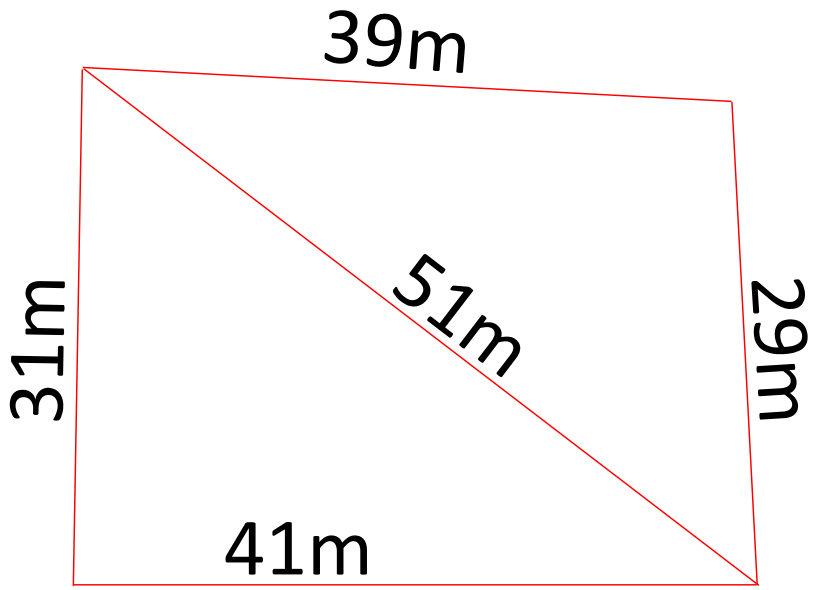
# 計算機構成論（おまけ）

## CとPythonの比較
## 例題３：関数の定義と呼び出し
## 例題４：ローカル変数
## 例題５：グローバル変数

2023年度前期
情報理工学部 Rクラス担当
越智裕之

# 例題３：土地の面積を計算せよ




幅40m×高さ30mの長方形で近似すれば、面積は約1200m²と概算できるが、正確な面積を知りたい。

【ヘロンの公式】

3辺の長さが $a, b, c$ である三角形の面積は

$$T = \sqrt{s(s-a)(s-b)(s-c)}$$

で与えられる。但し、

$$s = \frac{a+b+c}{2}$$

とする。

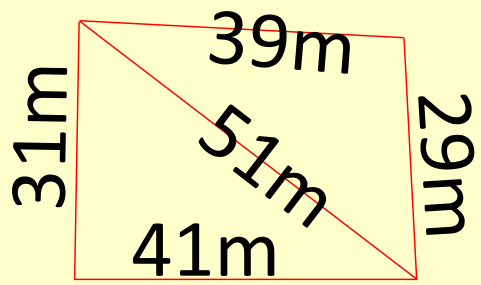
# 例題３：土地の面積を計算せよ
## （１）Python言語の場合

関数 heron() の定義

```
def heron(a,b,c):
    s = (a+b+c)/2.0
    t = (s*(s-a)*(s-b)*(s-c))**0.5
    return t
```

$s = \frac{a+b+c}{2}$の計算

$t = \sqrt{s(s-a)(s-b)(s-c)}$の計算




メインのプログラム

```
t1 = heron(31.0,41.0,51.0)
t2 = heron(29.0,39.0,51.0)
print t1+t2
```

関数 heron() 使って左下の三角形の面積を求める

関数 heron() 使って右上の三角形の面積を求める

２つの面積の和を表示する

# Pythonを知らない人のための補足説明

関数の定義は **def** で始める（返り値の型は書かない）

仮引数を括弧で括って列挙（引数の型は書かない）

関数名

コロン

平方根は **0.5

関数の中身は字下げ

```
def heron(a,b,c):
    s = (a+b+c)/2.0
    t = (s*(s-a)*(s-b)*(s-c))**0.5
    return t

t1 = heron(31.0,41.0,51.0)
t2 = heron(29.0,39.0,51.0)
print t1+t2
```

# 例題３：土地の面積を計算せよ
# （２）c言語の場合

関数 heron() の定義

メインのプログラム

```
#include <stdio.h>
#include <math.h>
double heron( double a, double b, double c ) {
   double s, t;
   s = (a+b+c)/2.0;
   t = sqrt(s*(s-a)*(s-b)*(s-c));
   return t;
}

int main( void ) {
   double t1, t2;
   t1 = heron(31.0,41.0,51.0);
   t2 = heron(29.0,39.0,51.0);
   printf("%f¥n",t1+t2);
}
```

$s = \frac{a+b+c}{2}$の計算

$t = \sqrt{s(s-a)(s-b)(s-c)}$の計算

関数 heron() 使って左下の三角形の面積を求める

関数 heron() 使って右上の三角形の面積を求める

２つの面積の和を表示する

# Cを知らない人のための補足説明

```
#include <stdio.h>
#include <math.h>

double heron( double a, double b, double c ) {
   double s, t;
   s = (a+b+c)/2.0;
   t = sqrt(s*(s-a)*(s-b)*(s-c));
   return t;
}
int main( void ) {
   double t1, t2;
   t1 = heron(31.0,41.0,51.0);
   t2 = heron(29.0,39.0,51.0);
   printf("%f¥n",t1+t2);
}
```

常に必要な呪文

**sqrt()** を使うときの呪文

関数名

仮引数を括弧で括って列挙
（引数の型を書き添える！）

中括弧

返り値の型

変数の宣言

中括弧

平方根は **sqrt()** 関数で計算

変数の宣言

メインプログラムは **main** という名前の関数

中括弧

# 例題３：土地の面積を計算せよ ソースコードの比較

- Pythonの方が簡潔だが、基本的には似ている

```c
#include <stdio.h>
#include <math.h>
double heron( double a, double b, double c ) {
  double s, t;
  s = (a+b+c)/2.0;
  t = sqrt(s*(s-a)*(s-b)*(s-c));
  return t;
}
int main( void ) {
  double t1, t2;
  t1 = heron(31.0,41.0,51.0);
  t2 = heron(29.0,39.0,51.0);
  printf("%f¥n",t1+t2);
}
```

C言語（14行）

```python
def heron(a,b,c):
  s = (a+b+c)/2.0
  t = (s*(s-a)*(s-b)*(s-c))**0.5
  return t
t1 = heron(31.0,41.0,51.0)
t2 = heron(29.0,39.0,51.0)
print t1+t2
```

Python言語（7行）

# 例題３：土地の面積を計算せよ 実行方法の比較

- Cはコンパイルが必要

```
% gedit area.c
% gcc -lm -o area area.c
% ./area
1197.751920
%
```

ソースファイルの編集（拡張子は .c）

コンパイラ gcc を使い実行形式ファイルを生成（**sqrt()** を使うときは **-lm** オプションが必要）

実行

printf 文の表示

- Pythonはすぐに実行できる

```
% gedit area.py
% python3 area.py
1197.75192012
%
```

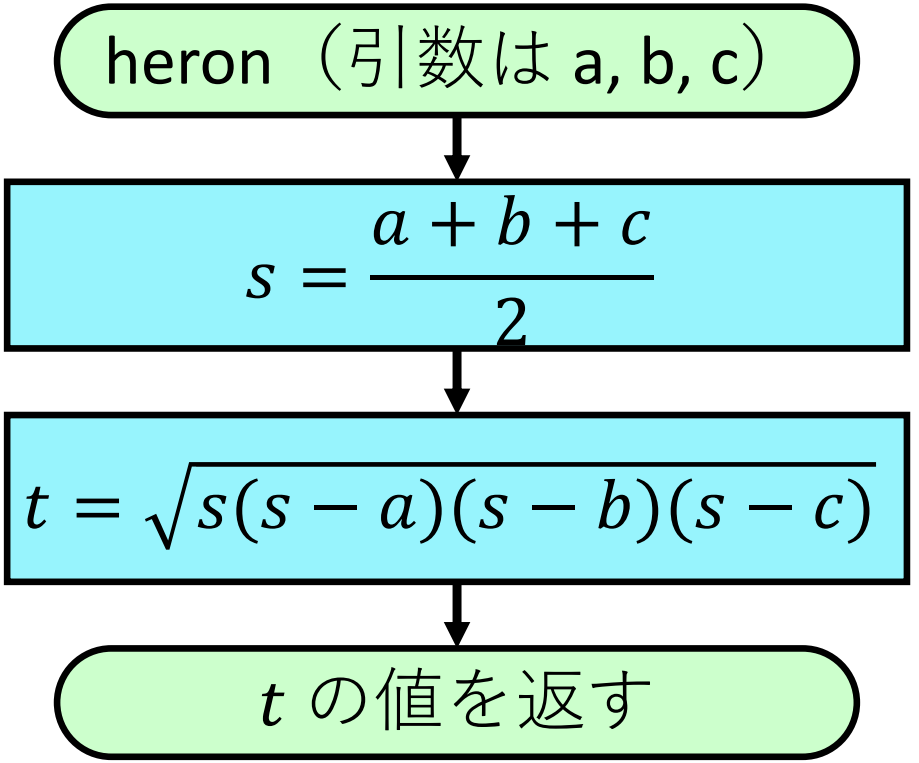
ソースファイルの編集（拡張子は .py）

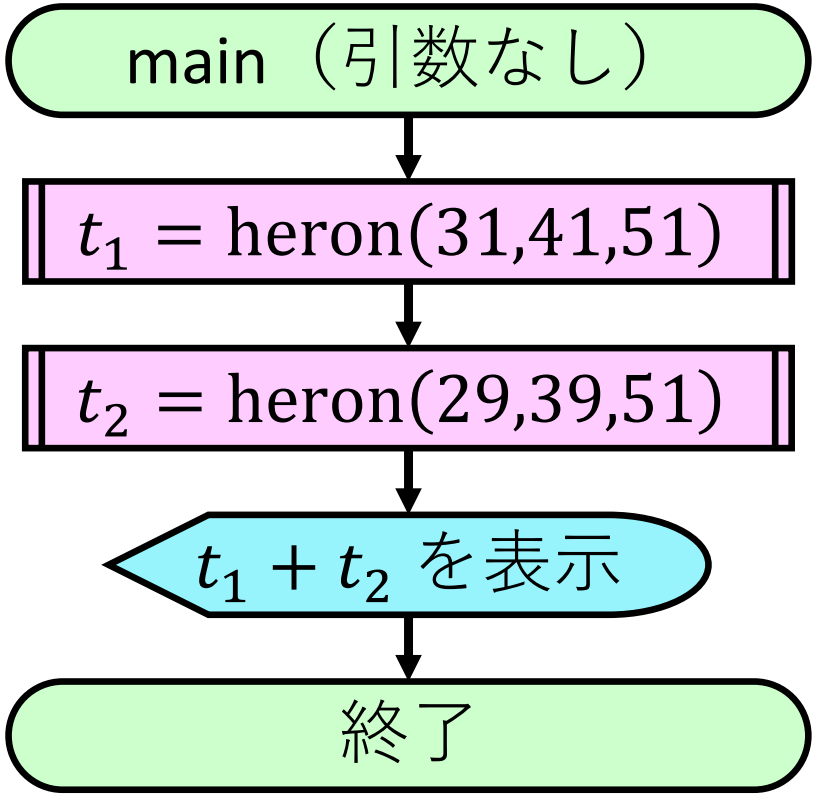
python3コマンドにソースファイルを読ませることで実行

print 文の表示

# フローチャートとの対応

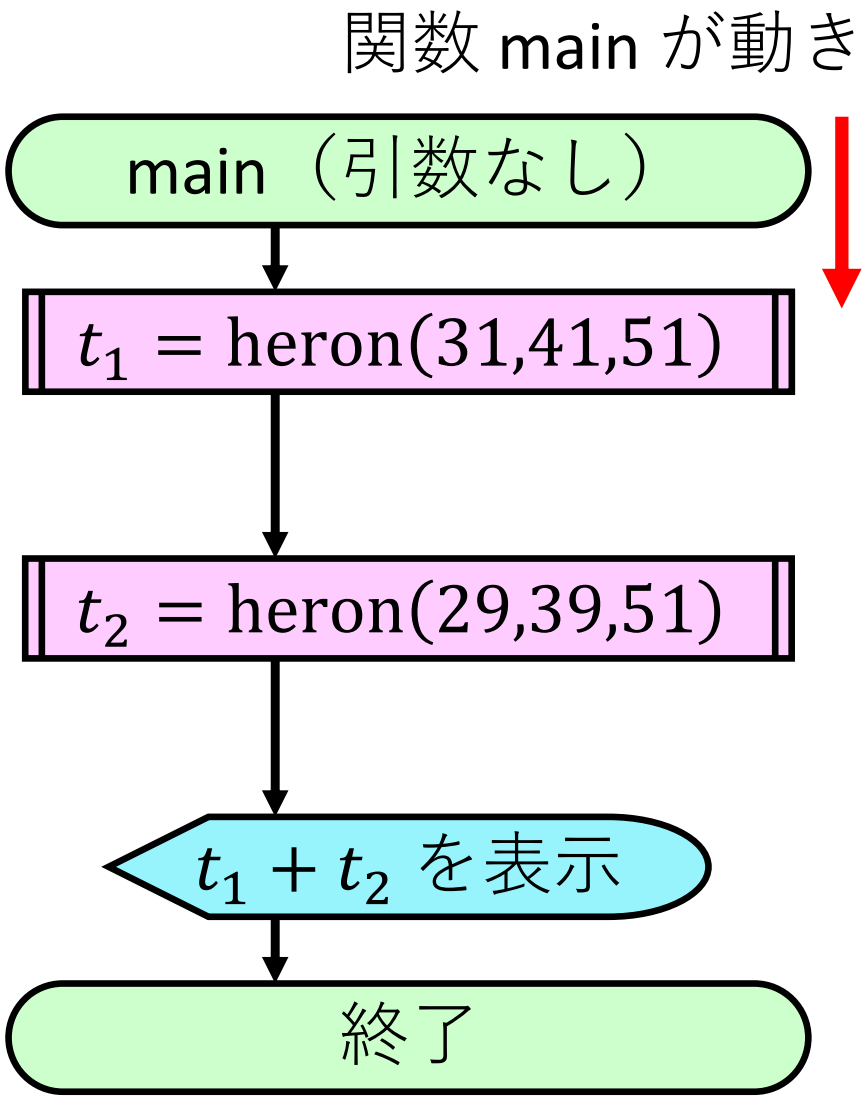
```
double heron( double a, double b, double c ) {
  double s, t;
  s = (a+b+c)/2.0;
  t = sqrt(s*(s-a)*(s-b)*(s-c));
  return t;
}
```

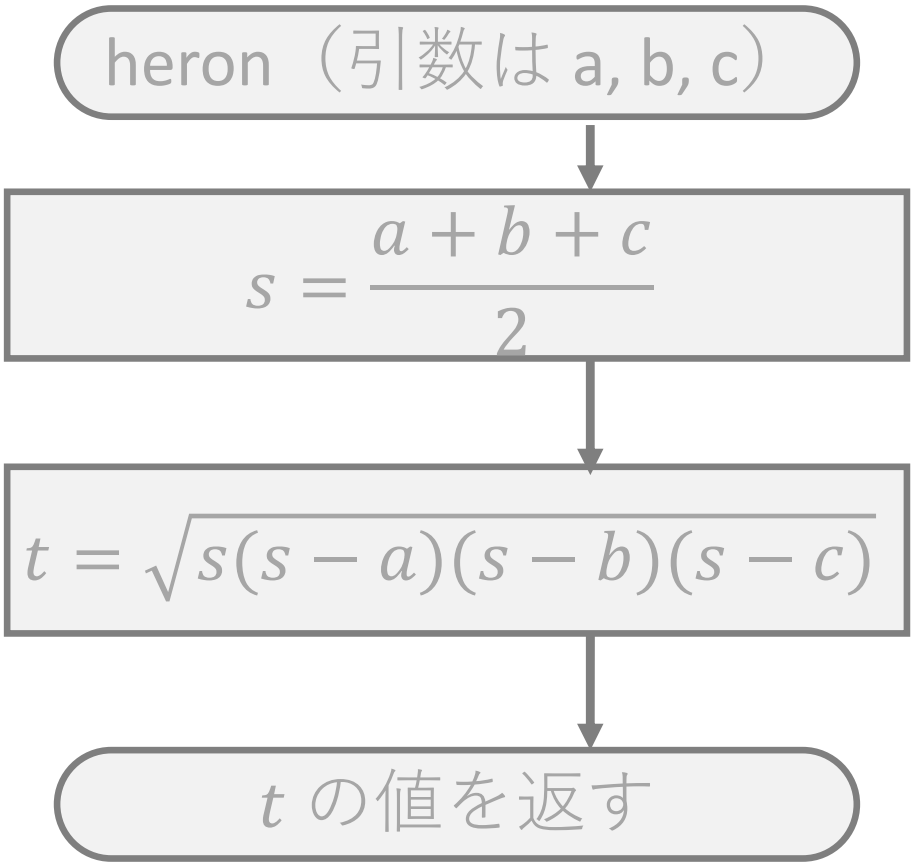



```
int main( void ) {
  double t1, t2;
  t1 = heron(31.0,41.0,51.0);
  t2 = heron(29.0,39.0,51.0);
  printf("%f¥n",t1+t2);
}
```

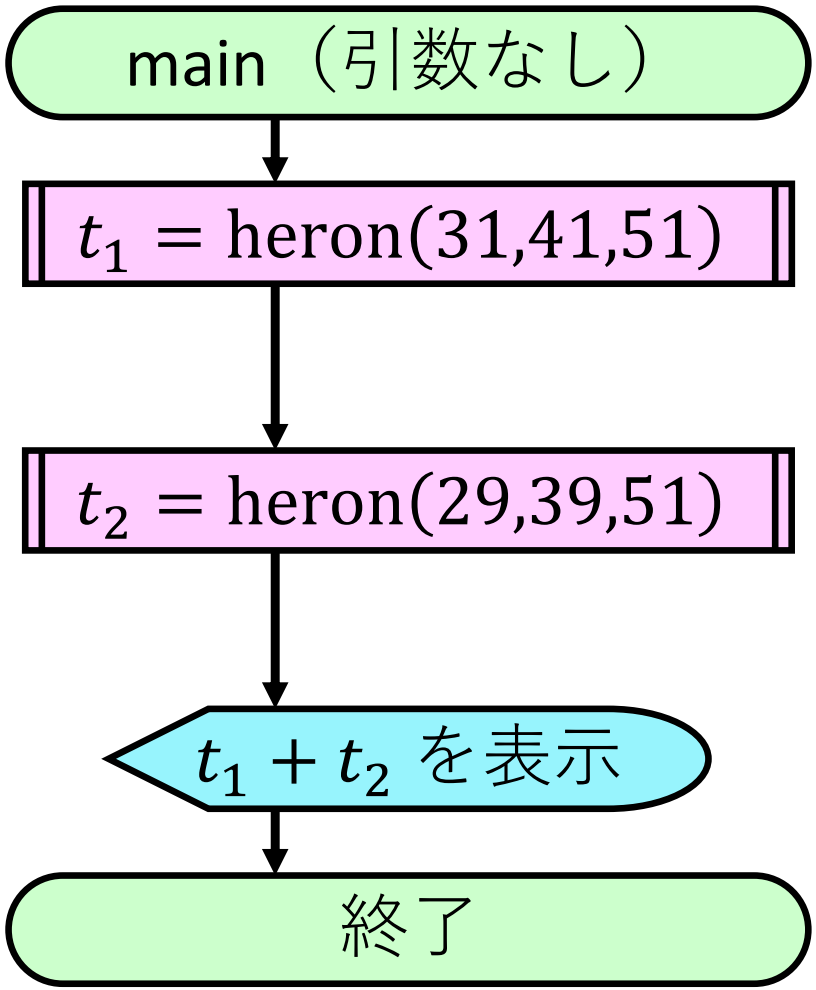
# 処理の流れ（１）

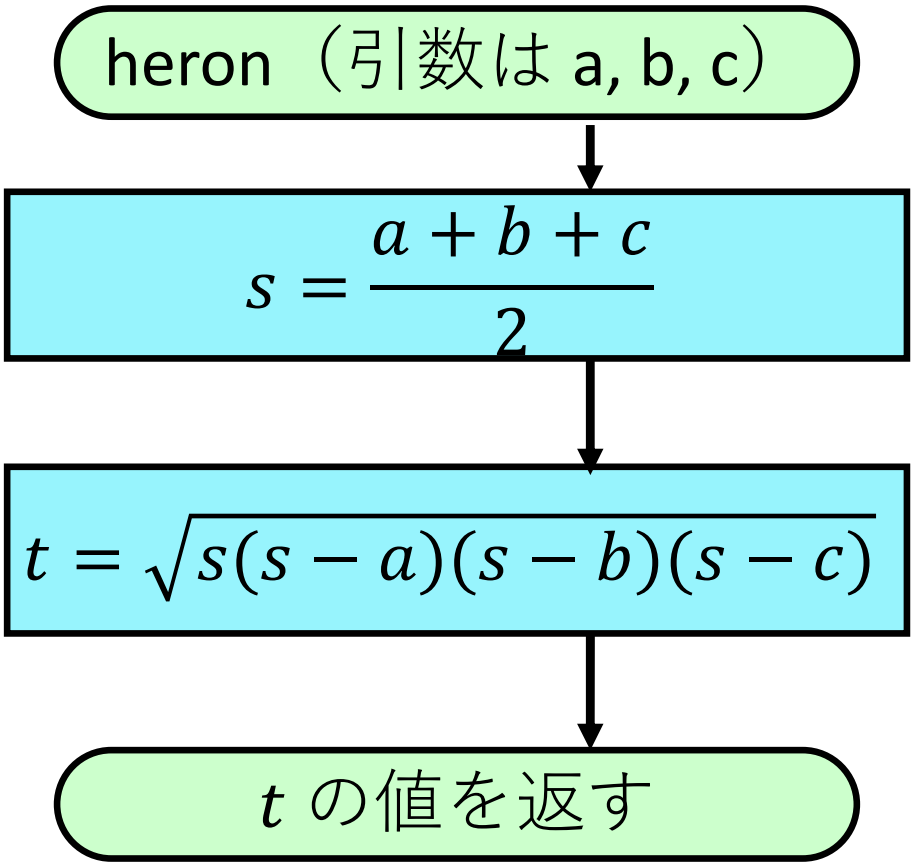

関数 main が動き始める
main（引数なし）
$t_1 = \mathrm{heron}(31,41,51)$
$t_2 = \mathrm{heron}(29,39,51)$
$t_1 + t_2$ を表示
終了


heron（引数は a, b, c）
$s = \frac{a+b+c}{2}$
$t = \sqrt{s(s-a)(s-b)(s-c)}$
$t$ の値を返す

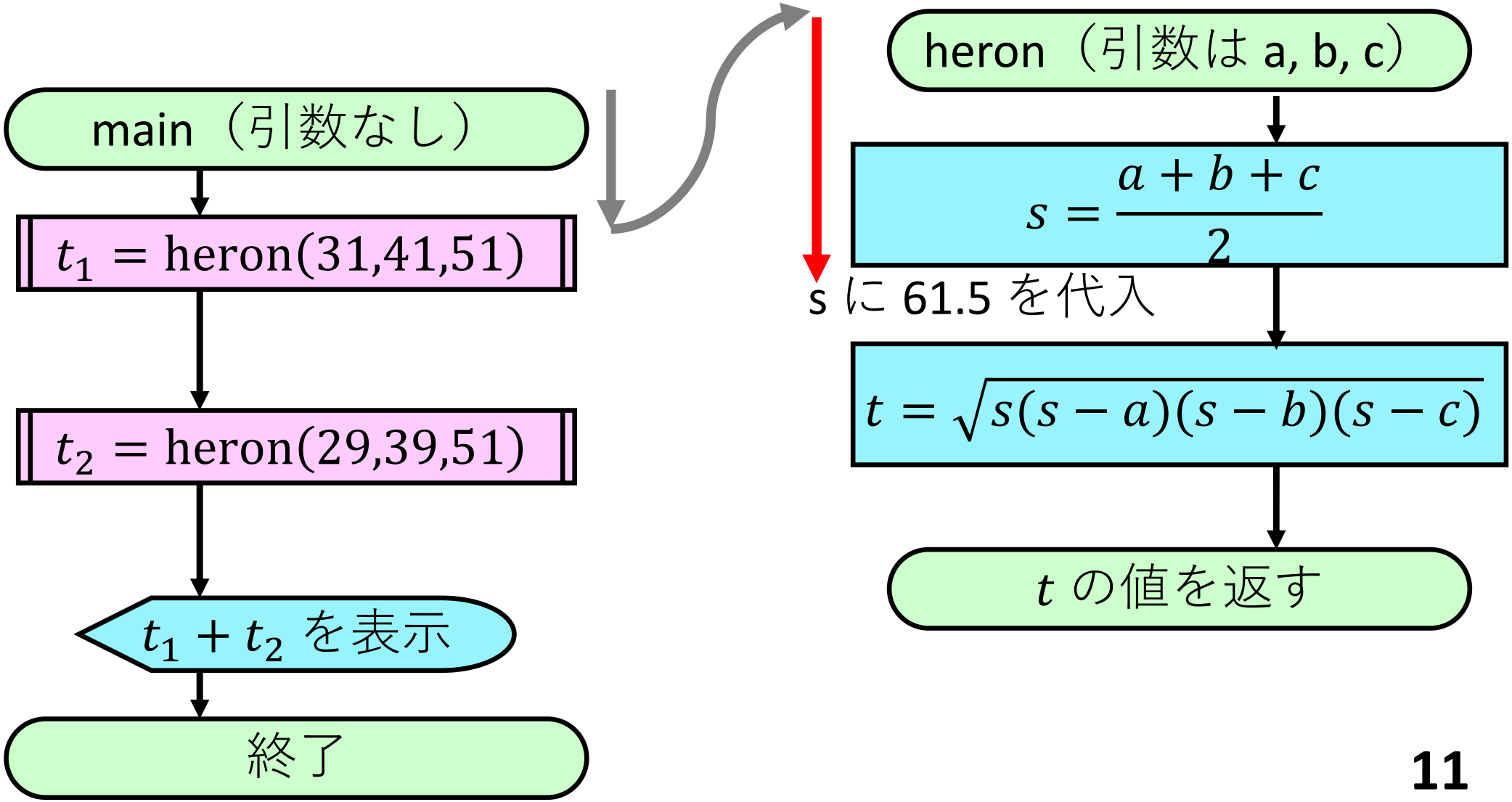
# 処理の流れ（2）

- 関数 heron() のローカル変数 a, b, c, s, t のメモリ領域を確保
- a に 31.0 を代入
- b に 41.0 を代入
- c に 51.0 を代入

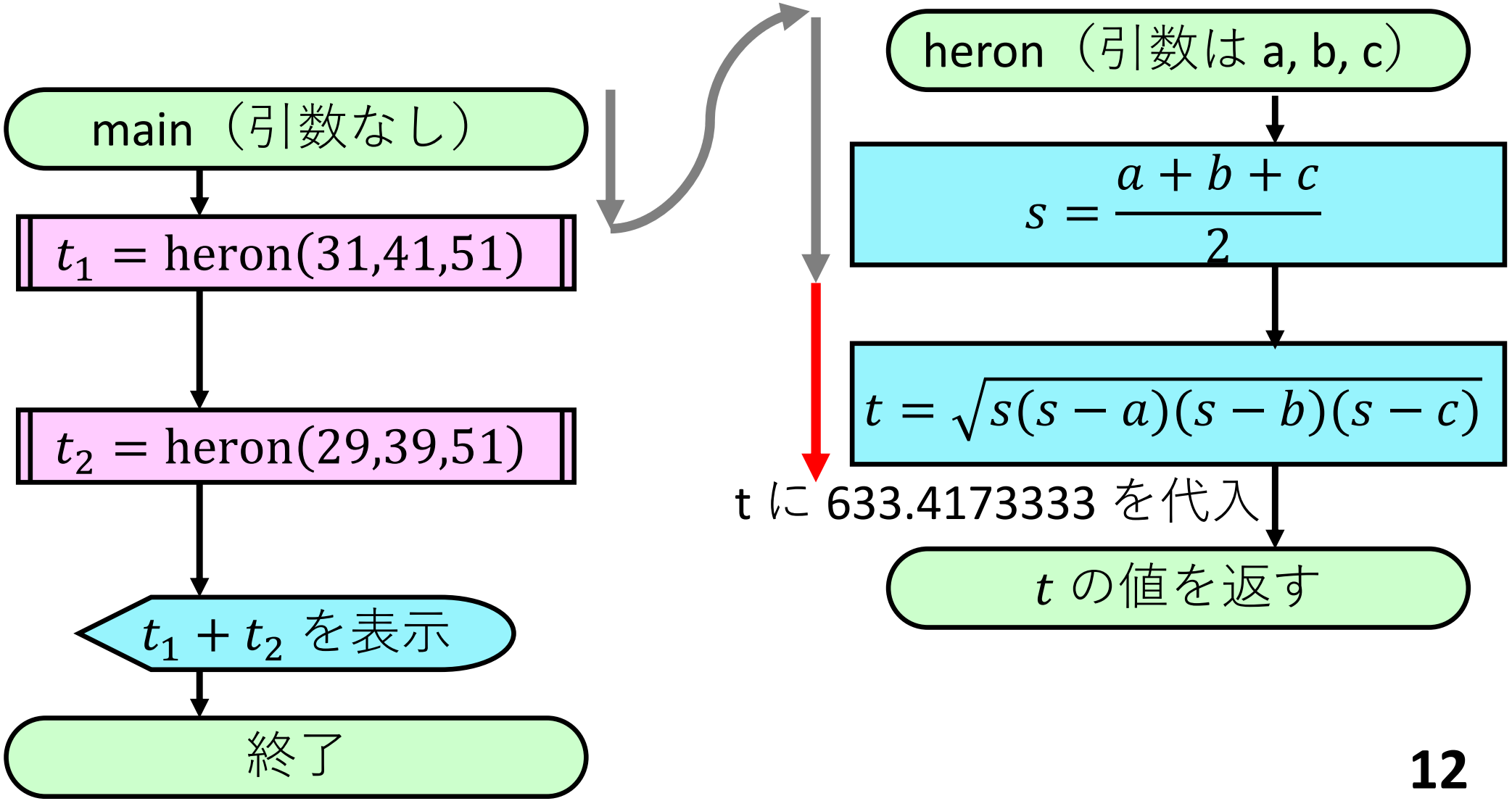


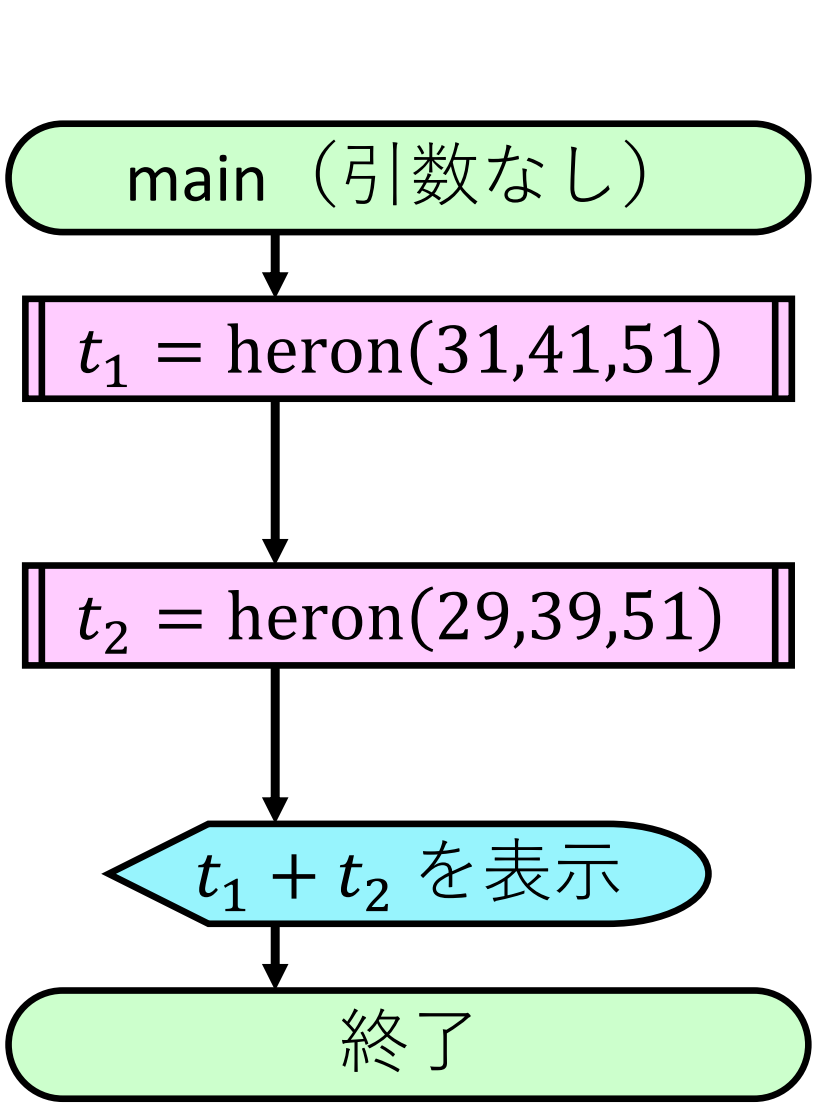
# 処理の流れ（３）

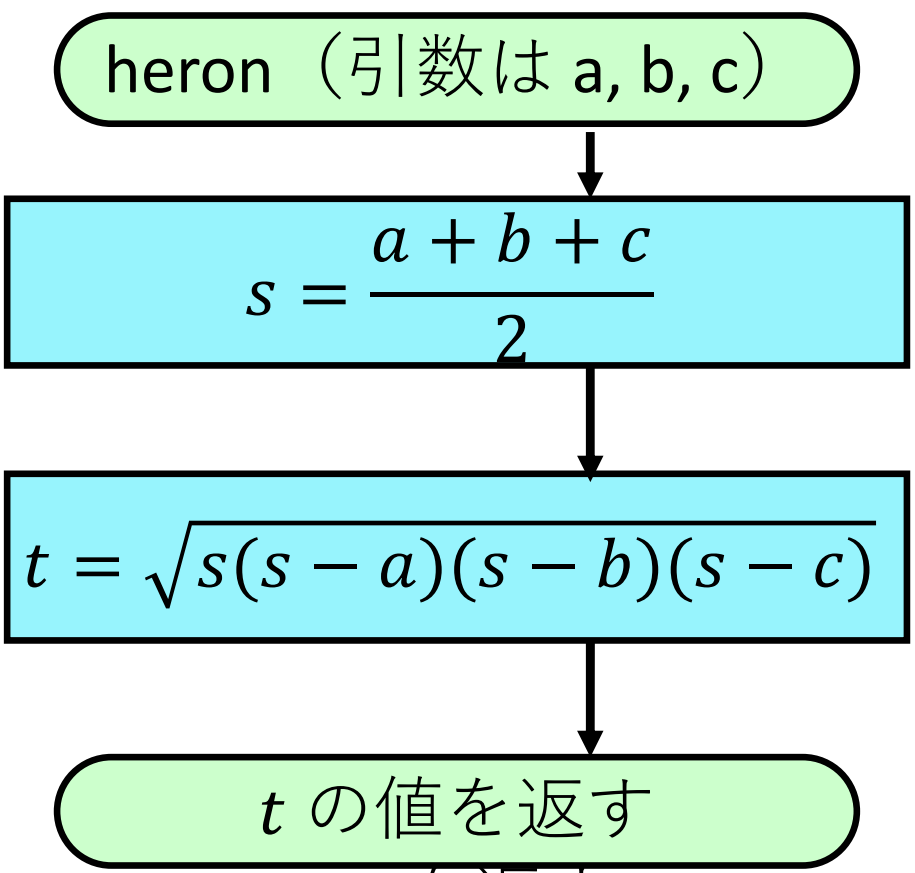

main（引数なし）
$t_1 = \mathrm{heron}(31,41,51)$
$t_2 = \mathrm{heron}(29,39,51)$
$t_1 + t_2$ を表示
終了
heron（引数は a, b, c）
$s = \frac{a+b+c}{2}$
s に 61.5 を代入
$t = \sqrt{s(s-a)(s-b)(s-c)}$
$t$ の値を返す

# 処理の流れ（４）


main（引数なし）
$t_1 = \mathrm{heron}(31,41,51)$
$t_2 = \mathrm{heron}(29,39,51)$
$t_1 + t_2$ を表示
終了
heron（引数は a, b, c）
$s = \frac{a+b+c}{2}$
$t = \sqrt{s(s-a)(s-b)(s-c)}$
t に 633.4173333 を代入
$t$ の値を返す

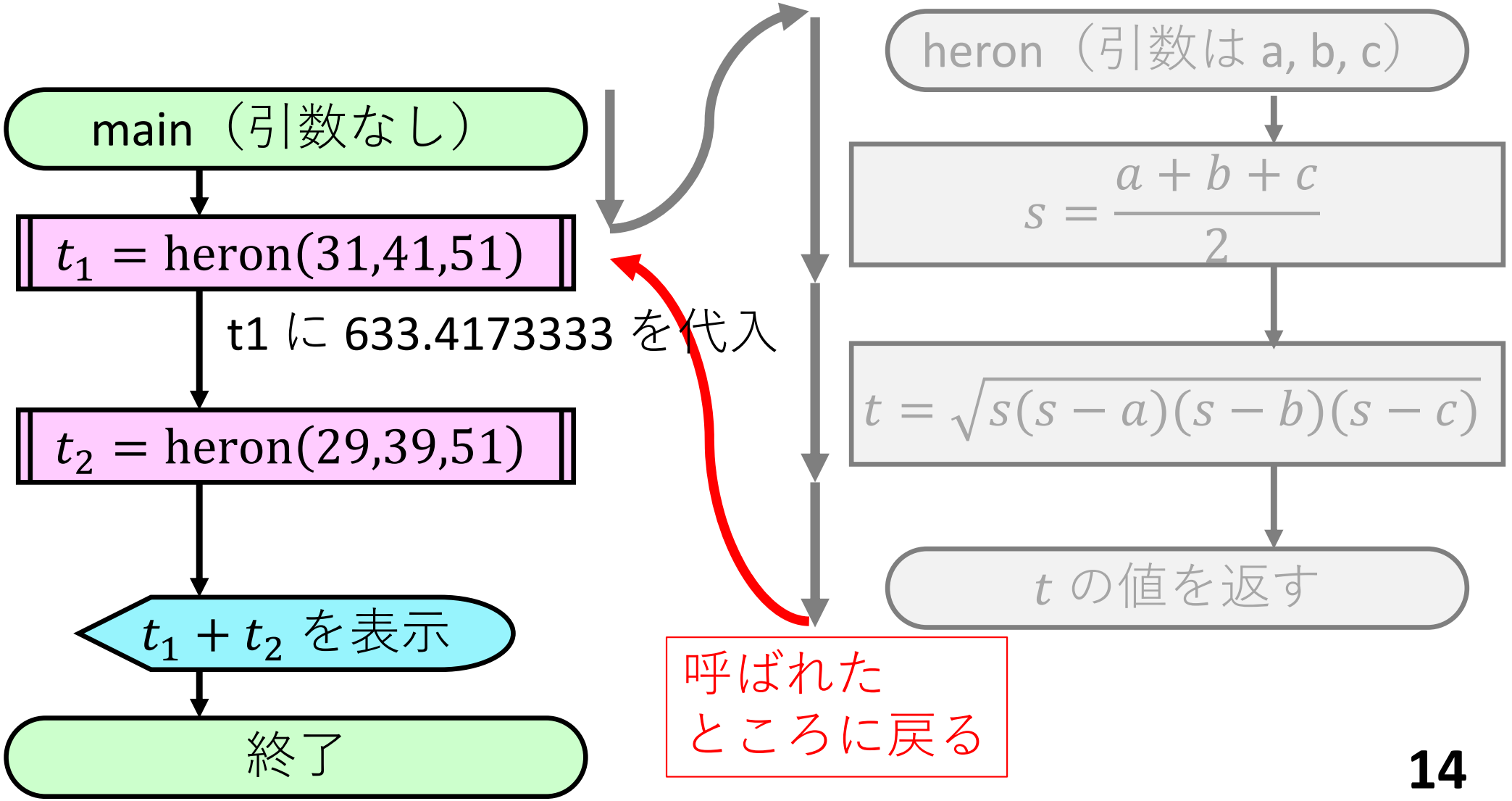
# 処理の流れ（５）



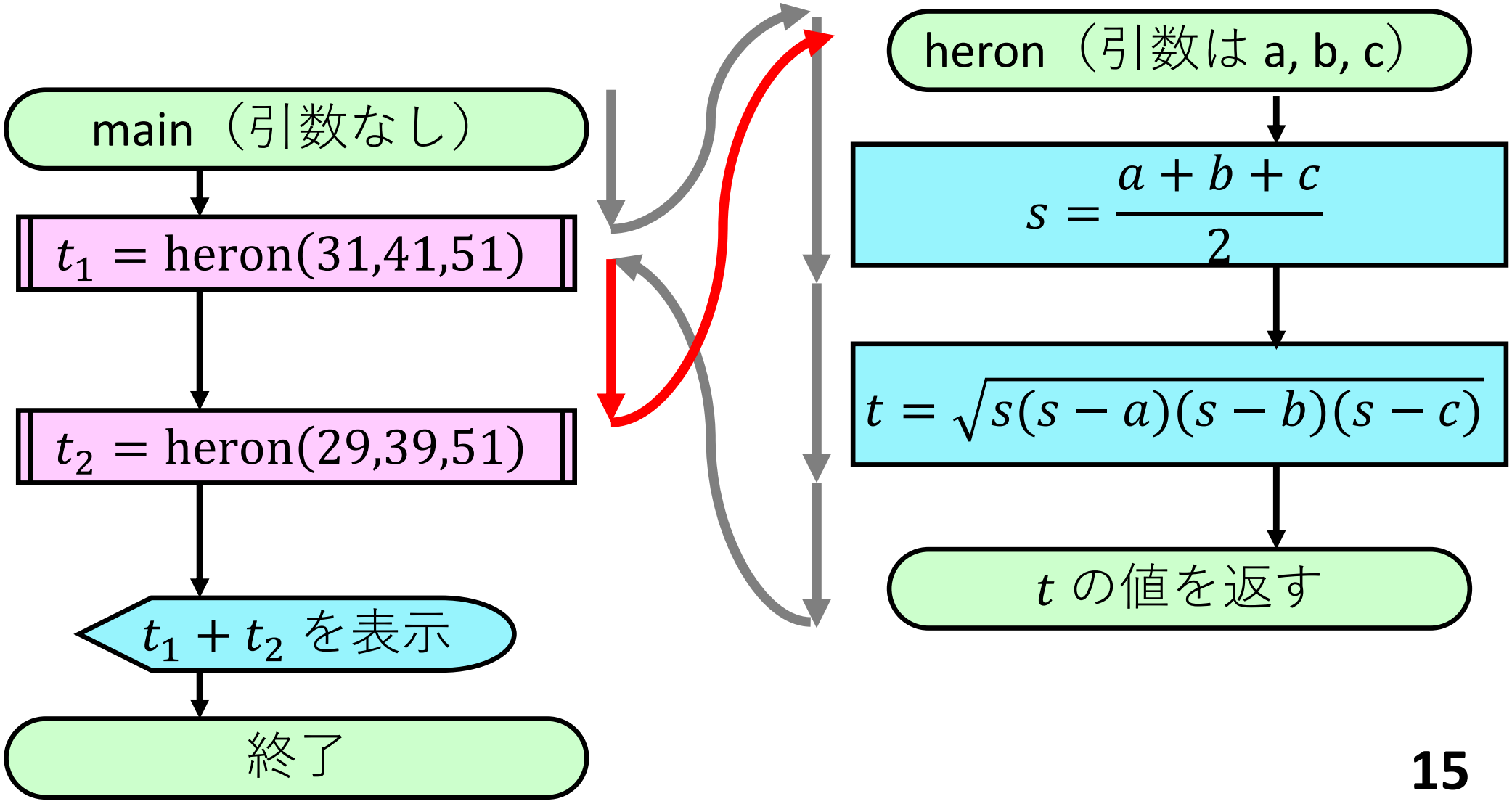

- 633.4173333 を返す
- 関数 heron() のローカル変数 a, b, c, s, t のメモリ領域を破棄

# 処理の流れ（6）

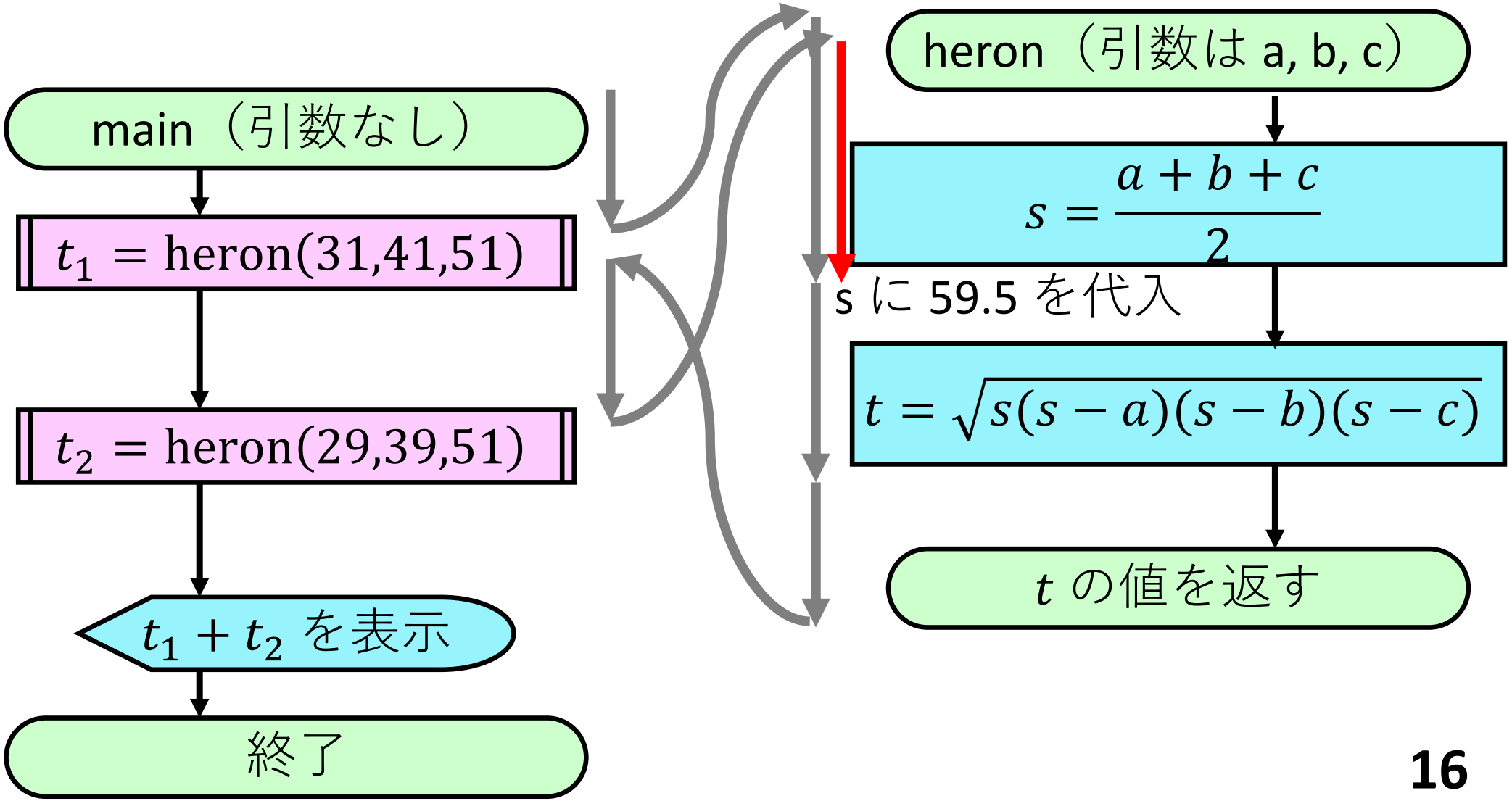

main（引数なし）
$t_1 = \mathrm{heron}(31,41,51)$
t1 に 633.4173333 を代入
$t_2 = \mathrm{heron}(29,39,51)$
$t_1 + t_2$ を表示
終了
heron（引数は a, b, c）
$s = \frac{a+b+c}{2}$
$t = \sqrt{s(s-a)(s-b)(s-c)}$
$t$ の値を返す
呼ばれた
ところに戻る

# 処理の流れ（７）

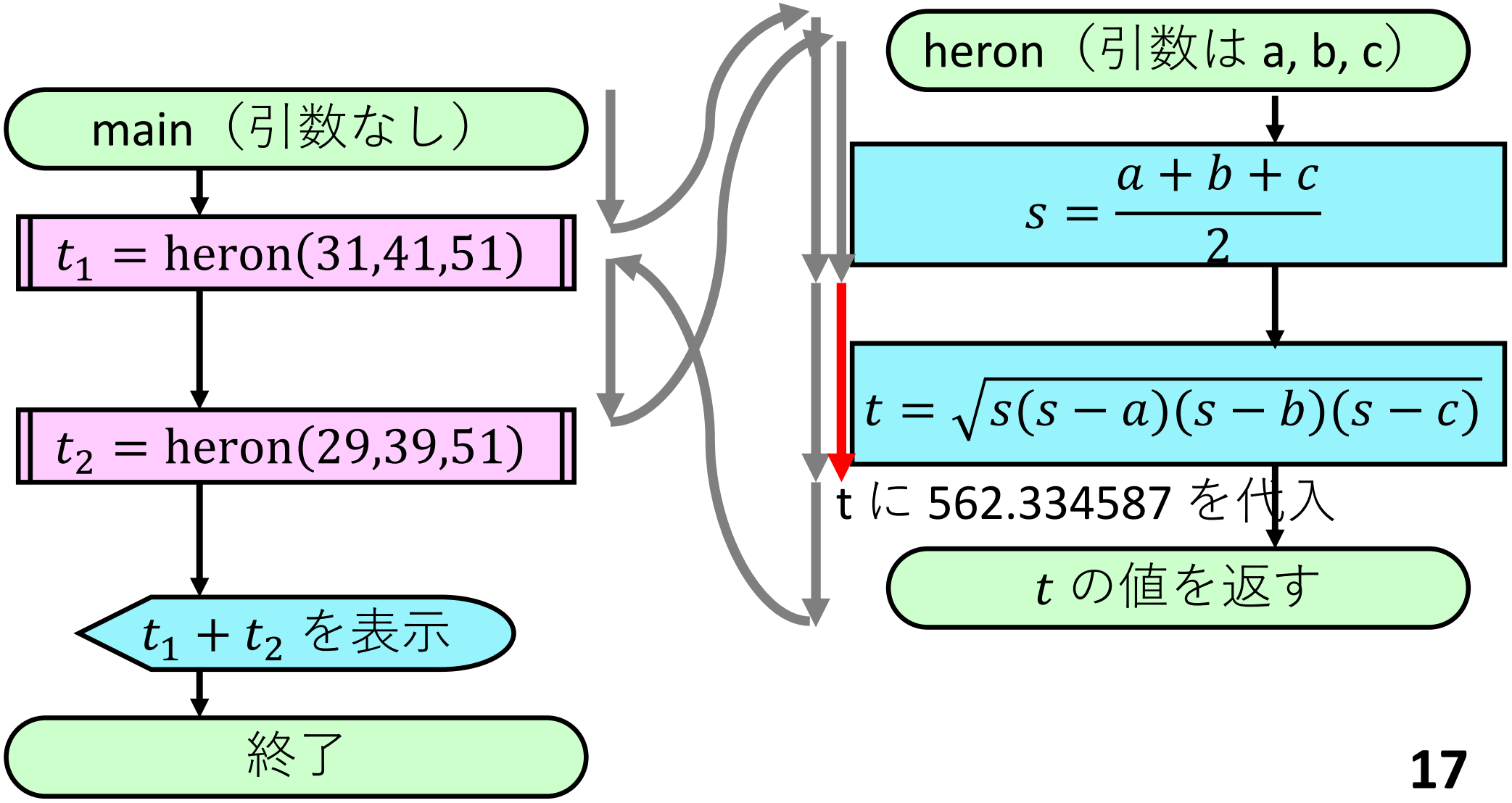
- 関数 heron() のローカル変数 a, b, c, s, t のメモリ領域を確保
- a に 29.0 を代入
- b に 39.0 を代入
- c に 51.0 を代入



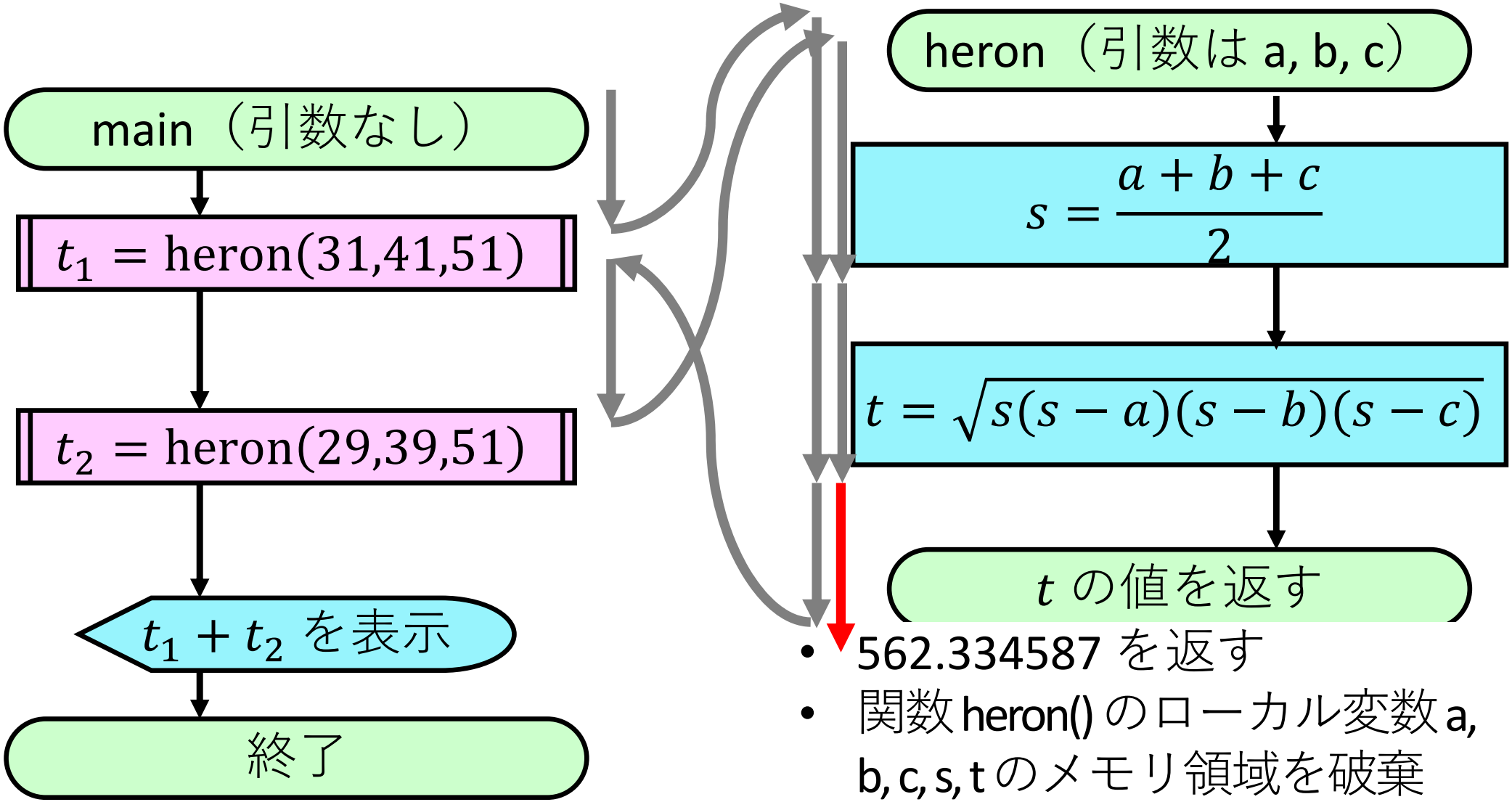
# 処理の流れ（8）


main（引数なし）
$t_1 = \mathrm{heron}(31,41,51)$
$t_2 = \mathrm{heron}(29,39,51)$
$t_1 + t_2$ を表示
終了
heron（引数は a, b, c）
$s = \frac{a+b+c}{2}$
s に 59.5 を代入
$t = \sqrt{s(s-a)(s-b)(s-c)}$
$t$ の値を返す

# 処理の流れ（９）

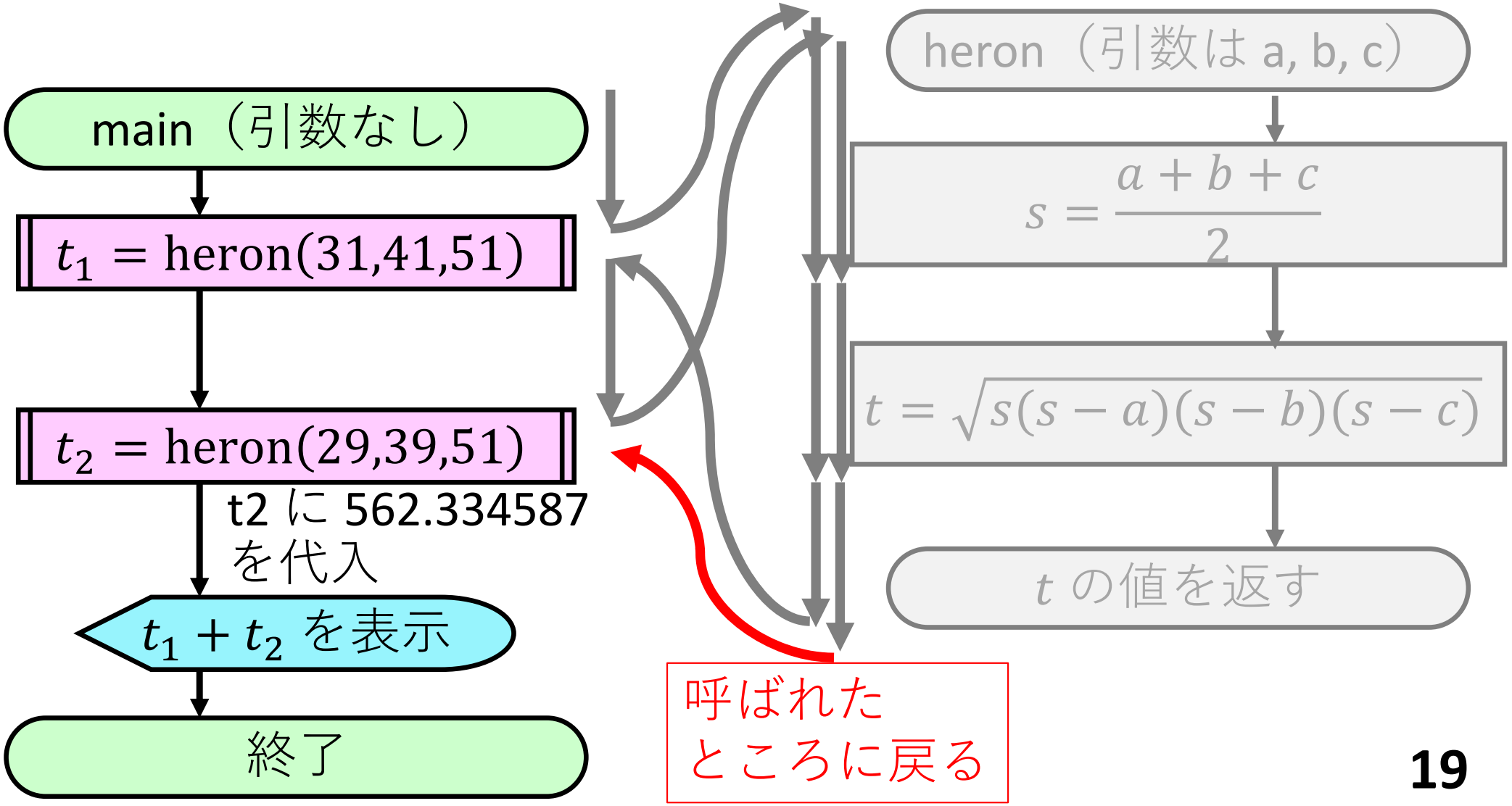

main（引数なし）
$t_1 = \mathrm{heron}(31,41,51)$
$t_2 = \mathrm{heron}(29,39,51)$
$t_1 + t_2$ を表示
終了
heron（引数は a, b, c）
$s = \frac{a+b+c}{2}$
$t = \sqrt{s(s-a)(s-b)(s-c)}$
t に 562.334587 を代入
$t$ の値を返す

# 処理の流れ（１０）

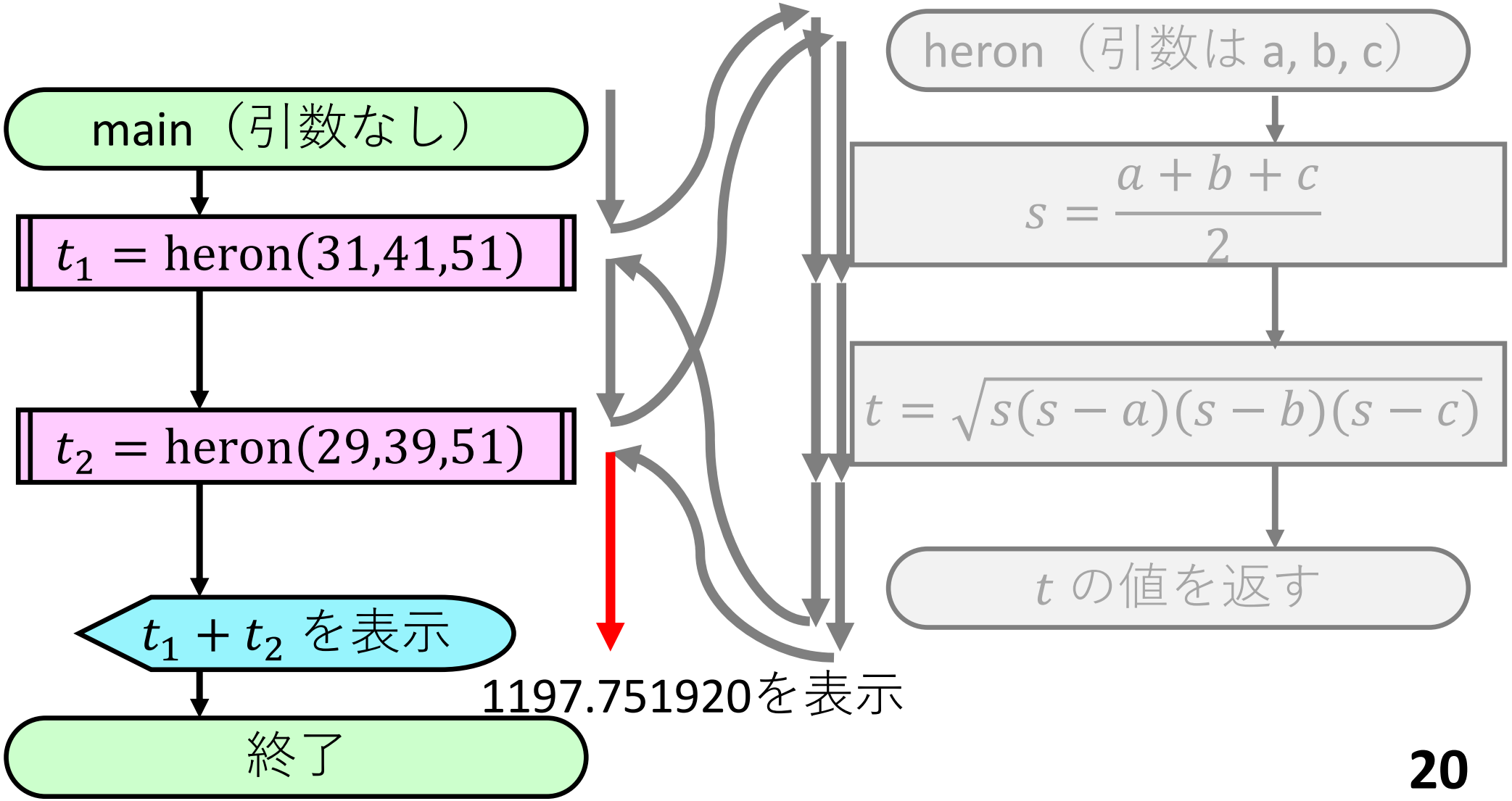



- 562.334587 を返す
- 関数 heron() のローカル変数 a, b, c, s, t のメモリ領域を破棄

# 処理の流れ（１１）


main（引数なし）
$t_1 = \mathrm{heron}(31,41,51)$
$t_2 = \mathrm{heron}(29,39,51)$
t2 に 562.334587 を代入
$t_1 + t_2$ を表示
終了
heron（引数は a, b, c）
$s = \frac{a+b+c}{2}$
$t = \sqrt{s(s-a)(s-b)(s-c)}$
$t$ の値を返す
呼ばれた ところに戻る

# 処理の流れ（１２）


main（引数なし）
$t_1 = \mathrm{heron}(31,41,51)$
$t_2 = \mathrm{heron}(29,39,51)$
$t_1 + t_2$ を表示
終了
heron（引数は a, b, c）
$s = \frac{a+b+c}{2}$
$t = \sqrt{s(s-a)(s-b)(s-c)}$
$t$ の値を返す
1197.751920を表示

# 例題４：関数内で定義したローカル変数
# （１）Python言語で書いた階乗の関数

n の階乗を計算して返す関数 factorial() の定義

```
def factorial(n) :
    f = 1
    i = 1
    while i <= n :
        f *= i
        i += 1
    return f

print factorial(5)
```

関数 **factorial()** 内で使われている変数 **n** や **f** や **i** は、この関数内でのみ有効（ローカル変数）

# 例題４：関数内で定義したローカル変数
# （２）Python言語で書いた階乗の関数の応用

```
def factorial(n) :
    f = 1
    i = 1
    while i <= n :
        f *= i
        i += 1
    return f


i = 1
while i <= 10 :
    print factorial(i)
    i += 1
```

関数外で使われる変数 i と関数内で使われる i は、**名前は同じでも別の変数**

- 関数が呼び出されて関数内で i が変更されても関数外の i は破壊されない
- メインルーチンのプログラミング中に関数内のローカル変数を意識しなくて良い

# 例題４：関数内で定義したローカル変数
# （３）C言語で書いた階乗の関数

```
#include <stdio.h>

int factorial(int n) {
   int f = 1;
   int i = 1;
   while (i <= n) {
      f *= i;
      i += 1;
   }
   return f;
}

int main( void ) {
   int i = 1;
   while (i <= 10) {
      printf("%d¥n", factorial(i));
      i += 1;
   }
}
```

C言語でも、関数外で使われる変数 i と関数内で使われる i は、**名前は同じでも別の変数**

# 例題５：関数間で共有されるグローバル変数
## （１）Python言語で書いた失敗例

変数 bank に引数 money を足す関数 deposit() の定義

変数bankから引数 money を引く関数 draw()の定義

```
def deposit(money) :
    bank += money
    return

def draw(money) :
    bank -= money
    return

bank = 0          # 残高ゼロで開始
deposit(500)      # 500円貯金
draw(150)         # 150円引き出し
print bank        # 残高は？
```

関数 **deposit()** や **draw()** 内で使われている変数 **bank** は、これらの関数のローカル変数
（メインルーチンの変数 **bank** とは別物）

# 例題５：関数間で共有されるグローバル変数
# （２）Python言語で期待通り動くプログラム

変数 bank に引数 money を足す関数 deposit() の定義

変数bankから引数 money を引く関数 draw()の定義

```
def deposit(money) :
    global bank
    bank += money
    return

def draw(money) :
    global bank
    bank -= money
    return

bank = 0          # 残高ゼロで開始
deposit(500)      # 500円貯金
draw(150)         # 150円引き出し
print bank        # 残高は？
```

関数 **deposit()** や **draw()** 内でメインルーチンの変数 **bank** をアクセスしたいときは global 宣言を書く

「350」と表示される

# 例題５：関数間で共有されるグローバル変数
# （３）C言語で書いた失敗例

```
#include <stdio.h>

void deposit(int money) {
  bank += money;
  return;
}

void draw(int money) {
  bank -= money;
  return;
}

int main( void ) {
  int bank;
  bank = 0;                  // 残高ゼロで開始
  deposit(500);              // 500円貯金
  draw(150);                 // 150円引き出し
  printf("%d¥n",bank);       // 残高は？
}
```

関数 **main()** 内で宣言された変数 **bank** を他の関数 **deposit()** や **draw()** 内からアクセスすることはできない

# 例題５：関数間で共有されるグローバル変数
# （４）C言語で期待通り動くプログラム

```
#include <stdio.h>

int bank;

void deposit(int money) {
   bank += money;
   return;
}

void draw(int money) {
   bank -= money;
   return;
}

int main( void ) {
   bank = 0;                  // 残高ゼロで開始
   deposit(500);              // 500円貯金
   draw(150);                 // 150円引き出し
   printf("%d¥n",bank);       // 残高は？
}
```

C言語の場合、このように関数定義の外で宣言された変数は**グローバル変数**となり、どの関数からもアクセスできる

「350」と表示される