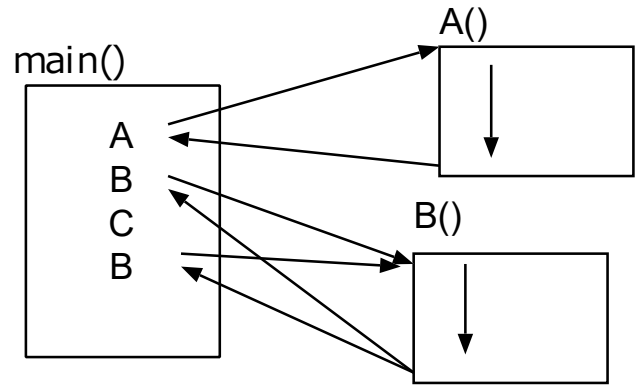
C++プログラミングI

- 第5回 関数定義とスコープ
- 担当：二瓶芙巳雄

関数とは

- 一連の処理に名前をつけたもの
- プログラムを区切って読みやすくする
- 用途：
  - 似たような処理を一つにまとめる
  - 一般的な関数を作り再利用（ライブラリ関数）
  - プログラムの管理を関数ごとに



関数の呼び出しと，処理の流れのイメージ図

補足：身近な関数

- 三角関数：$\sin(\cdot)$，$\cos(\cdot)$，$\tan(\cdot)$
  - $\sin(45^\circ) = \frac{1}{\sqrt{2}} = 0.70710$
- `std::sin()`は，`<cmath>`に定義された**関数**
  - `std::sin()`に角度（**引数**）を与えて，計算結果（**戻り値**）を得ている

```
1  #include <cmath>
2
3  int main() {
4    double value { std::sin(45) };
5    std::cout << value << "\n";
6
7    std::cout << std::cos(45) << "\n";
8  }
```

※`std::sin()`は引数として弧度法の値を受け取るので，上のプログラムは期待通りの値になりません

補足：関数にまとめる

- 三角形の面積：$\frac{w \cdot h}{2}$
  - w, hはそれぞれ三角形の底辺の長さと高さ
- 右上のコードでは，面積を求める式をベタ書き
  - 同じ処理をまとめると再利用性が高い
  - 公式を知らない人は式の意味がわからない
    - 三角形の面積の公式を知らない人はいませんが…
- 右下のコードでは，面積の計算を自作の関数で
  - 同じ処理を2度書かないで済む
  - 処理に名前を与えられる

```
int main() { // 愚か
  double w1{2}, h1{3};
  double scale1 { 0.5 * w1 * h1 };

  double w2{7}, h2{5};
  double scale2 { 0.5 * w2 * h2 };
}
```

```
double triangle_scale( double w, double h ) {
  double scale { 0.5 * w * h };
  return scale;
}

int main() { // 賢い
  double w1{2}, h1{3};
  double scale1 { triangle_scale(w1, h1) };

  double w2{7}, h2{5};
  double scale2 { triangle_scale(w2, h2) };
}
```

関数定義

- 戻り値の型、関数名、仮引数
  - **仮引数**: 呼び出される関数側の引数
  - **実引数**: 呼び出し側で指定する引数
- 値渡し：実引数の値が仮引数に渡される（代入）
  - ※値のコピーが渡されています
- **戻り値**：`return` 文で関数が返す値を指定
  - 戻り値は呼出し側で変数に代入できる
  - 関数呼び出しは式を構成する変数と同じ場所で指定

```cpp
// from から to までの整数値の合計を求める関数
int sumup(int from, int to) { // ← 仮引数
  int sum {0};
  for (int i = from; i <= to; i++) sum += i;
  return sum;
}

int main() {
  int s1;
  s1 = sumup(1, 10); // ← 実引数
  std::cout << s1 << "\n";
  int s2 { sumup(s1, s1+10) };
  std::cout << s2 << "\n";

  std::cout << sumup(s2, s2+10) << "\n";
  return 0;
}
```

`void` 関数

- 戻り値のない関数には `void` を指定する
  - 予約語 `void` は「無効」を表す
- 値指定なしの `return` を使う（任意）
- 戻り値がない関数は，直感的には「手続き」

```
% ./a.out
0
0
```

```
1  void print(int x) {
2     if (x == 0) return;
3     std::cout << x <<"\n";
4  }
5
6  int main() {
7    int x {0};
8    print( x );
9
10   int y {1};
11   print( y );
12
13   print( 0 );
14
15   return 0;
16 }
```

関数の引数

- 関数の引数は0個でも複数でも良い
- 引数0個の場合も丸括弧 `()` が必要
- 同一型の複数引数でもそれぞれに型指定が必要
  - ○：`int x, int y`
  - ×：`int x, y`
- 実引数には代入可能な式（右辺値）を指定する

```
1  int zero() { /*処理*/ }
2  int one(int x) { /*処理*/ }
3  int two(int x, int y) { /*処理*/ }
4  double dist(double x, double y) { /*処理*/ }
5  int mix(int x, double d) { /*処理*/ }
6
7  int main() {
8    std::cout << zero();
9    std::cout << one( 100 );
10   std::cout << two( 2, 3 );
11   std::cout << dist( 1.414, 3.141 );
12   std::cout << mix( 7, 5.28 );
13 }
```

補足：戻り値

- 関数が戻り値を持つとき，呼び出した関数は値として使用できる
- 関数が戻り値を持たないとき，呼び出した関数は値として使用できない
- `return`があるから値を戻すのではなく，`void`以外の戻り値を設定すると，`return`を書くことが期待される
  - pythonとはちょっと違う考え方

```
1   int func_i() { /*処理内容*/ }
2   double func_d( double w ) { /*処理内容*/ }
3   std::string func_s() { /*処理内容*/ }
4   void func_v() { /*処理内容*/ }
5
6   int main() {
7     int x { func_i() }; // OK
8     std::cout << x << func_d( 3.0 ) << "\n"; // OK
9     std::cout << "hello" + func_s() << "\n"; // OK
10
11    func_v(); // OK
12    std::cout << func_v() <<  "\n"; // エラー，voidは値ではない
13    void z { func_v() }; // エラー，変数の型としてvoidは不適
14    std::cout << 3 * func_v() << "\n"; // エラー，voidは演算ができない
15  }
```

- 戻り値を有する関数に`return`を書かないと`warning: no return statement in function returning non-void`
- `return`を書かない状態で戻り値を有する関数を呼び出すとランタイムエラー`Illegal instruction (core dumped)`

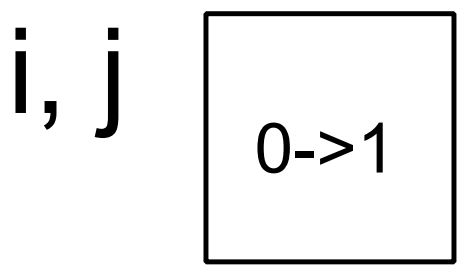
pythonと比較

```cpp
double half( int val ) {
   double h { val / 2.0 };
   return h;
}

int add( int a, int b ) {
   return a + b;
}

void print_value( double a ) {
   std::cout << "value is: " << a << "\n";
}

int main() {
  double half_value { half(5.0) };
  std::cout << half_value << " " << half(7.5) << "\n";

  std::cout << add( 3, 5 ) << "\n";

  print_value( 7.0 );
}
```

```python
def half( val ):
  h = val / 2.0
  return h


def add( a, b ):
  return a + b


def print_value( a ):
  print( "value is: ", a )


if __name__ == ""
  half_value = half( 5.0 )
  print( half_value, half( 7.5 ) )

  print( add( 3, 5 ) )

  print_value( 7.0 )
```

lvalueリファレンス

- **リファレンス**：参照によってある変数の別名となる
- リファレンスはlvalue(左辺値）となれる変数が対象
- 一度初期化したら他の変数に変更できない



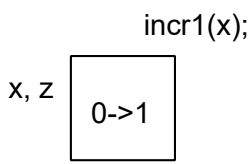

```cpp
int main() {
  int i{0};  // i は左辺値にできる
  int& j{i}; // j はi の別名
  ++j;
  std::cout << i <<" "<< j <<"\n";
}
```

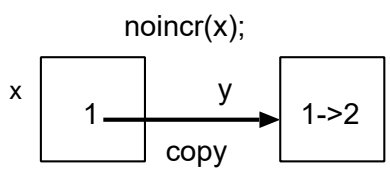
```
% ./a.out
1 1
```

`j`は`i`の別名なので，`j`に対する変更は`i`にも影響する

リファレンス引数

- lvalue リファレンスを引数に適用（**参照渡し**）
  - 関数内で実引数の値を直接変更する仕組み
- これまでの引数の例は**値渡し**
  - 関数の引数（仮引数）は実引数の値のコピー
  - 仮引数の値を変更しても、呼び出し側の実引数の変数には影響なし






```
1  void incr1(int &z) { ++z; }
2  void noincr(int y) { ++y; }
3
4  int main() {
5    int x{1};
6    incr1(x);  //x の値が更新される
7    noincr(x); //x は変更なし
8
9    std::cout << x <<"\n";
10 }
```

```
% ./a.out
2
```

変数の入れ替え関数(swap)

- 2つの実引数の値を入れ替える
- 引数にリファレンス `&` 指定がないと期待通りに動かない

```
1  void swap_int(int& a, int& b) {
2    int t {a};
3    a = b;
4    b = t;
5  }
6
7  int main() {
8    int x{10}, y{20};
9
10   swap_int(x,  y); // x とy の値の入れ替え
11   // std::swap(x, y); // ライブラリ関数がある
12
13   std::cout << x << " " << y << "\n";
14 }
```

`vector` の引数

- A: 値渡しの場合には全要素がコピーされる
  - 効率が悪い
- B: リファレンス引数では要素はコピーされない
  - 効率が良い
- C: リファレンス引数だと実引数を変更できてしまう．変更できないように `const` を付ける
- 一般的には，BかCで指定するとよい
  - これは `vector` を含む，ユーザ定義型（クラス，構造体など）でも同様

```cpp
int sum_value(std::vector<int> a) { // A
  int s {0};
  for (int i : a) s += i;
  return s;
}

int sum_ref(std::vector<int>& a) { // B
  int s {0};
  for (int i : a) s += i;
  return s;
}

int sum_ref(const std::vector<int>& a) { // C
  int s {0};
  for (int i : a) s += i;
  return s;
}
```

`main` 関数の仮引数

- `main()` とは何なのか？
  - プログラムのエントリポイント
  - 関数である以上，引数と戻り値がある
  - 引数（**コマンド引数**）：黒い画面から渡す
  - 戻り値：黒い画面に戻す値
- コマンド引数を使うには？
  - 利用にはC配列とポインタの知識が必要
  - `argc`：コマンド引数の数,
  - `argv`：文字列へのポインタ配列
- 右でコマンド引数を `vector<string>` に変換できる
  - パターンなので覚えてしまいましょう

```cpp
int main( int argc, char *argv[] ) { // 引数省略なし

  // argv(コマンド引数)をvectorに取り込む
  std::vector<std::string> a(argv, argv+argc);

  for (int i = 0; i < a.size(); i++)
    std::cout << i << ": " << a[i] <<"\n";

  return 0; // main()の戻り値．どこに戻す？
}
```

```
% g++ -std=c++17 sample5.cpp
% ./a.out train leaves 7 a.m.
0: ./a.out
1: train
2: leaves
3: 7
4: a.m.
%
%
% # 豆知識：以下を入力すると，main()の戻り値を表示できます
% echo $?
0
```

局所変数のスコープ(有効範囲)

- 局所変数（関数の中の変数），有効範囲がある
- `{}` の範囲をブロックと呼ぶ
- ブロック内にそのブロック用の局所変数を宣言できる
  - `{}` の内側で宣言された変数は，その `{}` の外側からアクセスできない（※例外あり）
- ブロックは内部に別のブロックを持てる(入れ子構造)
- 内側のブロックの変数名が外側のものと同じならば、外側を一時的に隠す

```
1  void func() {
2    int x {1};    // 宣言a)
3    if (x > 0) {
4      int x {2};  // 宣言b) 宣言a) のx を隠す
5
6      // ここで表示される値は 2
7      std::cout << x <<"\n";
8    }
9
10   // 再び宣言a) のx が使える
11   // ここで表示される値は 1
12   std::cout << x <<"\n";
13 }
14
15 int main(int argc, char *argv[]) {
16   func();
17 }
```

例外的なルール

- 仮引数は関数本体のブロックがそのスコープ
- 制御文の文末までのスコープ
  - `for` 文の初期設定で宣言された変数
  - `if`, `while`, `switch` 文の条件部で宣言された変数
  - 複文でない場合は中括弧 `{}` がないので注意

```
1  for (int i = 0; i < 10; i++) {
2     cout << i <<"\n";
3  }
4  if (int n = abs(x)) {
5     cout << 10/n <<"\n";
6  }
7  while (char ch = s[i++]) {
8     cout << ch <<"\n";
9  }
```

スコープの利用

- 変数のスコープを限定して読みやすいコード作成を!

```
1  int count1() { // cntが重要であることが明確
2     int cnt {0};
3     for (int i = 1; i <= 100; i++) {
4        if (int x {(i*i) % 3}; x == 0)
5          ++ cnt;
6     }
7     return cnt;
8  }
9  int count2() { // 局所変数の用途が明確でない例
10    int i, x, cnt = 0;      // 3個の変数が同等
11    for (i = 1; i <= 100; i++) {
12       x = (i*i) % 3;
13       if (x == 0) ++ cnt;
14    }
15    return cnt;
16 }
```

同じ名前の変数

- スコープが異なれば変数に同じ名前をつけても良い

```
1  void func1() {
2     int x{};        // func1 に局所的な変数 x
3     x = 1;          // funcの局所変数を更新
4  }
5  void func2(int x){ // func2に局所的な仮引数x
6     x = 2;          // 仮引数xを更新
7  }
8  int func3(int x){  // 仮引数 xはfunc3に局所的
9     return x+1;     // 戻り値
10 }
11 int main() {
12    int x {0};      // main に局所的な変数 x
13    func1();
14    func2(x+1);     // x+1 は実引数
15    x = func3(2);   // 2 は実引数
16 }
```

大域変数

- 大域変数：関数間で共通に使える変数
  - 宣言した場所より下側関数で利用できる
  - 以下では `sum`, `from`, `to`
- 局所変数：関数の中で宣言された変数
  - 宣言した場所の関数でだけ利用できる
  - 仮引数も局所変数

```cpp
#include <iostream>

// 大域変数の宣言
int sum {0};
int from{0}, to{10};

void sumup() {
  sum = 0;  // 大域変数sum の利用
  for (int i = from; i <= to; i++)
    sum += i;
}

int main() {
  sumup();
  std::cout << sum << "\n";
}
```

変数の名前づけの慣習

- 局所変数には短めの名前
  - 使用範囲が限定的のため
- 大域変数には長めの名前
  - 使用範囲が広いので混乱を避ける目的

変数の初期値

- 初期値を指定しない組み込み型の変数
  - 大域変数は実行開始までに0に初期化される
  - 仮引数以外の局所変数の初期値は不定
  - 仮引数は実引数で初期値が指定される
- ユーザ定義型の変数の初期値はそれぞれで異なる
  - `string` 型の変数, `cin` , `cout` など

```
1  int    globalint;
2  double globalreal;
3  string globalstr;
4
5  void func(int p1, double p2) {
6     int    l1;            // 値は不定
7     double l2;            // 値は不定
8     string l3;            // 空文字列で初期化
9  }
```

大域変数と局所変数の関係

- 大域変数と局所変数が同じ名前ならば局所変数が優先
- 同じ名前とならないようにする
- 大域変数はスコープ解決演算子 `::` で明示可能

```cpp
int x {0}; // 大域変数x 初期値0

void func1() {
  x = 1;   // 大域変数x に代入
}
void func2() {
  int x{}; // func2 に局所的な変数x
  x = 2;   // 局所変数x に代入
}
void func3() {
  int x{}; // func3 に局所的な変数x
  ::x = 3; // 大域変数x に代入
}
int main() {
  std::cout << x << "\n"; // 0

  func1();
  std::cout << x << "\n"; // 1

  func2();
  std::cout << x << "\n"; // 1

  func3();
  std::cout << x << "\n"; // 3
}
```

宣言と条件(C++17)

- `if` と `switch` の変数宣言と条件指定．複雑な論理式が指定できる．
- セミコロンで宣言と論理式を区切る

if文

```
1   int main() {
2     int z{ std::min(x, y) };
3     if (z > 10)
4       std::cout << z << " is too large\n";
5
6     // ↑ を ↓ に書き換え可能
7
8     if (int z{ std::min(x, y) }; z > 10)
9       std::cout << z << " is too large\n";
10  }
```

switch文

```
1   int main() {
2     int z{ std::min(x, y) * 3 };
3     switch (z) {
4       case 3: std::cout << "three\n"; break;
5       case 6: std::cout << "six\n";   break;
6       case 9: std::cout << "nine\n";  break;
7     }
8     // ↑ を ↓ に書き換え可能
9     switch (int z{ std::max(x, y) }; z*3) {
10      case 3: std::cout << "three\n"; break;
11      case 6: std::cout << "six\n";   break;
12      case 9: std::cout << "nine\n";  break;
13    }
14  }
```