C++プログラミングI

第 10 回 関数の名前

成蹊大学理工学部

関数を使う意義

プログラムの処理を整理する

- まとまった処理に名前をつける
  - プログラム全体が 100 行を越えたら区分け
- プログラムの階層的な構成
  - main 関数に概要となる関数名が並ぶ
  - 個々の関数でその処理方法を明示
  - さらに下請けの関数を使うこともある
  - 文章の段落、節、章の構成と同じ考え
- 重複した処理の記述を減らす
  - たとえ数行でも何度も使う場合は更新忘れ防止になる
  - 類似処理は引数付きの関数にまとめる

関数宣言

- 関数の呼び出しには事前にその宣言が必要
- 関数宣言と関数定義
  - 関数宣言は関数名・戻り値の型・引数の数と型
  - 関数定義は関数宣言に関数本体の情報を加えたもの
- 細かなルール
  - 対応する関数の宣言と定義では、同じ型の指定が必要
  - 同じ情報の関数宣言は複数回書いても良い
  - 関数宣言の引数名は無視される

```
void incr(vector<int>& v, int x);// 引数名付き
void incr(vector<int>&, int);
void print(const vector<int>&);
```

関数の名前

- プログラムの読みやすさを決める重要な要素
- 適切な長さの名前が求められる
- 英語の動詞、特に単純なものは有用

悪い関数名の例：

```
foobar();
getUserInputValueFromKeyboard();
```

内容の想像がつく関数名の例：

```
void   print(const vector<int>&);
size_t find(const vector<string>&, string);
string get(istream&);
int count(const vector<Point>&, const Point&);
```

関数名の多重定義

- 引数が異なれば同名の別関数を定義してよい
  - 戻り値の型のみが異なる関数は定義できない
- オーバーロードとも呼ばれる
- 単純な動詞を関数名にし、引数を目的語にすると良い

```
// 判定方法の異なる 2 つの find 関数
 size_t find(const vector<int>&, int);
 size_t find(const vector<double>&, double);

// 入力を行う 2 つの get 関数
 bool get(int&, int& );
 bool get(vector<int>&);
```

const に注意

- const の有無で別関数が作れる

```
void print(const vector<int>& a) {
  cout <<"1\n";
}

void print(vector<int>& a) {
  cout <<"2\n";
}

int main() {
   const vector x {1,2,3,4,5};
   vector y {1,2,3,4,5};
   print(x);  // 1
   print(y);  // 2
}
```

デフォルト引数

- 関数呼び出し時の実引数の指定を省略する機能
  - 特殊な値の実引数を指定しデフォルト値は省略する
- 末尾の仮引数からデフォルト値を指定できる
  - 実引数も対応して末尾から省略する
- 関数宣言と定義のどちらかで 1 回のみ指定可能

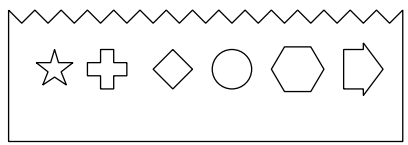
```
int sum(int, int, int =0, int =0); // 関数宣言
int multiply(int x,int y,int z =1){// 関数定義
  return x*y*z;
}

sum(1, 2, 3, 4);
sum(1, 2, 3);
sum(1, 2);
multiply(200, 300);
multiply(200, 300, 500);
```

関数テンプレート

- 変数の型のみが異なる別関数をまとめる
- 呼び出しの指定に応じて多重定義の関数ができる
- `template <typename T>`を関数の前に書く
  - `T`はテンプレート引数（名前は任意）
  - `template <class T>`と書くこともある
  - `template <class T, class K>`と複数でも良い

```
template <typename T>
size_t find(const vector<T>& a, T x)
{
   for (size_t i = 0; i < a.size(); i++)
      if (a[i] == x) return i;
   return a.size(); // 見つからない場合
}
```



関数テンプレートの使用

- テンプレート引数が推測可能な場合
- テンプレート引数の明示指定が必要な場合
- find 関数の第 1,3 引数の型に注意
  - "Nagoya"の型は char 型の配列
  - string 型は文字列リテラルを初期値に指定可

```
vector ia {3, 6, 2, 8, 1};
size_t n0 { find(ia, 8) };

vector<string> sa 略{......};
string city {"Fukuoka"};
size_t n1 { find(sa, city) };
size_t n2 { find(sa, string("Nagoya")) };
size_t n3 { find<string>(sa, "Osaka")  };
```

最小値を見つける関数テンプレート

```
template<class T>
T find_min(const vector<T>& a) {
   if (a.empty()) return T();
   T min { a[0] };
   for (size_t i = 1; i < a.size(); i++)
      if (a[i] < min) min = a[i];
   return min;
}
...
 vector x {3,5,2,4,7};
 cout << find_min(x) <<"\n";
 vector y {8.2, 3.4, 1.5, 6.3, 7.5};
 cout << find_min(y) <<"\n";
 vector<string> z {"FYI","EOM","PFA","BTW"};
 cout << find_min(z) << "\n";
```

- T() は型 T のデフォルト値を表す (ex. int なら 0)

便利な関数テンプレート

- <algorithm>ヘッダファイル

```
template<class T>
const T& min(const T& a, const T& b);

template<class T>
const T& max(const T& a, const T& b);

template<class T>
void swap(T& a, T& b);
```

s リテラル

- ユーザ定義リテラル (C++11)
- 標準ライブラリの string リテラル (C++14)
- サフィックス s（接尾辞）

```
using namespace std::string_literals;

vector sa {"Tokyo"s, "Sapporo"s};
size_t n2 = find(sa, "Nagoya"s);
```

関数名に関する注意

- 多重定義, デフォルト引数, 関数テンプレートのどれでも同名の別関数が作成できる
- 戻り値の型のみが異なる関数は作れない
- コンパイラは引数の型変換を試みるかもしれない
  - `bool, char`→`int`
  - `float`→`double`
  - `int`↔`double`
  - `int`→`unsigned int`

関数を作る際の指針

1. 基本的には関数にはそれぞれ別の名前をつける。
2. 関数の処理の仕方が異なり、引数の数や型が異なるが、概念的に同じ処理内容の関数ならば (特に単純な単語で表現できる場合) 関数名の多重定義とする。
3. 引数の数が異なるだけならばデフォルト引数とする。
4. 引数の型のみが異なり処理内容が同じならば関数テンプレートとする。

関数作成に関する指針

前提：長く使うプログラムは何度も読み返して修正する（文章作成と同じ）。

1. 基本的に関数にはそれぞれ別の名前をつける。
2. 関数の処理の仕方が異なり、引数の数や型が異なるが、概念的に同じ処理内容の関数ならば (特に単純な単語で表現できる場合) 関数名の多重定義とする。
3. 引数の数が異なるだけならばデフォルト引数とする。
4. 引数の型のみが異なり処理内容が同じならば関数テンプレートとする。