# C++プログラミングI

- 第10回：関数宣言, 関数名の多重定義, デフォルト引数, 関数テンプレート
- 担当：二瓶芙巳雄

# 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

- `main()`を最初の方に書きたい場合もある
  - **関数の宣言**だけ先に済ませることが可能
- **関数宣言**：関数名・戻り値の型・引数の数と型
- **関数定義**：関数宣言に関数本体の情報を実装
- 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
  - 宣言と定義で仮引数の名前が異なっていてもよい

```cpp
void incr(vector<int>&, int);   // 関数宣言①
void print(const vector<int>&); // 関数宣言②

int main() {
  vector<int> a { 1,2,3,4,5 };
  print(a);
  incr(a, 3);
  print(a);
}

void incr(vector<int>& v, int x) { // 関数定義①
  for (auto& e : v) e += x;
}

void print(const vector<int>& v) { // 関数定義②
  for (const auto& e : v) cout << e << " ";
  cout << "\n";
}
```

```cpp
// 細かなルール具体例：関数宣言①は，以下のように書くこともできる
void incr(vector<int>&, int);     // 引数名なし
void incr(vector<int>& v, int x); // 引数名付き
void incr(vector<int>& foo, int bar); // 定義と引数名が違う
```

# 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(); // 何がしたいかわからない
getUserInputValueFromKeyboard();  // 名前が長いかも
```

内容の想像がつく関数名の例：

```
void print(const vector<int>&);
// ↑ 配列の各要素を表示してくれそう

size_t find(const vector<string>&, string);
// ↑ 値（第2引数）が配列（第1引数）のどこにあるか教えてくれそう

string get(istream&);
// ↑ 入力ストリームから文字列型の値を取り出してくれそう

int count(const vector<Point>&, const Point&);
// ↑ 値（第2引数）が配列（第1引数）に何個あるか教えてくれそう
```

# 関数名の多重定義

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

```
% ./a.out
4
3
```

```cpp
// 多重定義！
// intのためのfind()と，doubleのためのfind()をそれぞれ定義
size_t find(const vector<int>&, int);
size_t find(const vector<double>&, double);

int main() {
  vector<int> ia {3, 6, 2, 8, 5};
  std::cout << find(ia, 5) << "\n";

  vector<double> da = {3.10, 6.21, 2.33, 3.66, 1.51};
  std::cout << find(da, 3.66) << "\n";
}

size_t find(const vector<int>& a, int 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
}

size_t find(const vector<double>& a, double x) {
  const double epsilon { 1.0e-2 };
  for (size_t i = 0; i < a.size(); i++)
    if (std::abs(a[i] - x) < epsilon) return i;
  return a.size(); // 見つからない場合
}
```

# 多重定義：いつ使う？①

- **多重定義はプログラムをきれいに書くテクニック**
- 多重定義しない（左）：似たような関数名が沢山で面倒，関数呼び出しが冗長
- 多重定義する（右）：プログラマーが覚える関数の種類が減る，関数呼び出しがシンプルに

```cpp
void print_add_int( int a, int b ) {
  std::cout << a + b << "\n";
}

void print_add_double( double a, double b ) {
  std::cout << a + b << "\n";
}

void print_add_string( string a, string b ) {
  std::cout << a + b << "\n";
}

int main() {
  print_add_int( 1, 2 );
  print_add_double( 1.1, 2.2 );
  print_add_string( "c++", "programming" );
}
```

```cpp
void print_add( int a, int b ) {
  std::cout << a + b << "\n";
}

void print_add( double a, double b ) {
  std::cout << a + b << "\n";
}

void print_add( string a, string b ) {
  std::cout << a + b << "\n";
}

int main() {
  print_add( 1, 2 );
  print_add( 1.1, 2.2 );
  print_add( "c++", "programming" );
}
```

# 多重定義：いつ使う？②

- 多重定義を使って，冗長なコードをきれいにしていきましょう

```
1  void print_vector_int( vector<int> v ) {
2    for( int e : v ) cout << e << " ";
3  }
4  void print_vector_double( vector<double> v ) {
5    for( double e : v ) cout << e << " ";
6  }
7  void print_vector_string( vector<string> v ) {
8    for( string e : v ) cout << e << " ";
9  }
10
11 int main() {
12   vector<int> i_vec { 1, 2, 3 };
13   vector<double> d_vec { 1.1, 2.2, 3.3 };
14   vector<string> s_vec { "c++", "programming", "!" };
15
16   print_vector_int( ivec );
17   print_vector_double( dvec );
18   print_vector_string( s_vec );
19 }
```

```
1  void print_vector( vector<int> v ) {
2    for( int e : v ) cout << e << " ";
3  }
4  void print_vector( vector<double> v ) {
5    for( double e : v ) cout << e << " ";
6  }
7  void print_vector( vector<string> v ) {
8    for( string e : v ) cout << e << " ";
9  }
10
11 int main() {
12   vector<int> i_vec { 1, 2, 3 };
13   vector<double> d_vec { 1.1, 2.2, 3.3 };
14   vector<string> s_vec { "c++", "programming", "!" };
15
16   print_vector( ivec );
17   print_vector( dvec );
18   print_vector( s_vec );
19 }
```

# `const` に注意

- `const` の有無で別関数が作れる

```
1  void print(const vector<int>& a) {
2    cout <<"1\n";
3  }
4
5  void print(vector<int>& a) {
6    cout <<"2\n";
7  }
8
9  int main() {
10    const vector x {1,2,3,4,5};
11    vector y {1,2,3,4,5};
12    print(x);  // 1
13    print(y);  // 2
14 }
```

※教科書での記載はなさそうですが把握しておきましょう

# 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

```
% ./a.out
10 6 3
60000 30000000
```

```
1  // 関数宣言でデフォルト引数の指定
2  int sum(int, int, int = 0, int = 0);
3  // int sum(int=0, int, int, int); // コンパイルエラー
4
5  // 関数定義でデフォルト引数の指定
6  int multiply(int x, int y, int z = 1) {
7    return x*y*z;
8  }
9
10 int main() {
11   cout << sum(1, 2, 3, 4) << ' '
12        << sum(1, 2, 3) << ' '  // 引数の一部を省略
13        << sum(1, 2) << '\n';  // 引数の一部を省略
14
15   cout << multiply(200, 300) << ' ' // 引数の一部を省略
16        << multiply(200, 300, 500) << '\n';
17 }
18
19 // 関数宣言のタイミングでデフォルト引数を指定したので，
20 // 関数定義の時はデフォルト引数を指定しない
21 int sum(int a, int b, int c, int d) {
22   return a + b + c + d;
23 }
```

# デフォルト引数：いつ使う？

- **デフォルト引数は，同じ機能を持つ関数を一つにするテクニック**
- デフォルト引数を使わない（左）：2つの値の和，3つの値の和，…のような関数が同時に存在．冗長．
- デフォルト引数を使う（右）：一つの関数で，左側すべてに対応できる

```
1   int add_two( int a, int b ) {
2     return a + b;
3   }
4
5   int add_three( int a, int b, int c ) {
6     return a + b + c;
7   }
8
9   int add_four( int a, int b, int c, int d ) {
10    return a + b + c + d;
11  }
12
13  int main() {
14    std::cout << add_two( 1, 2 ) << "\n";
15    std::cout << add_three( 1, 2, 3 ) << "\n";
16    std::cout << add_four( 1, 2, 3, 4 ) << "\n";
17  }
```

```
1   int add( int a, int b, int c=0, int d=0 ) {
2     return a + b + c + d;
3   }
4
5   int main() {
6     std::cout << add( 1, 2 ) << "\n";
7     std::cout << add( 1, 2, 3 ) << "\n";
8     std::cout << add( 1, 2, 3, 4 ) << "\n";
9   }
```

※デフォルトの値は，関数の処理内容に応じて，よく考えて決めましょう．総和の場合は0が妥当ですが，例えば総積の場合は1が妥当です．

# 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）
- 複数種類のテンプレート引数を使いたいときは，`template <typename T, typename K>` のように書く
- `template <class T>` と書くこともある

関数を変更する際に、一つのコードを修正するだけで複数の型に対応した関数の修正が可能

```cpp
// 配列の中から指定要素を探し、配列の添字を返す関数テンプレート
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();
}

int main() {
  vector<int> ia {3, 6, 2, 8, 1};
  cout << find(ia, 8) << "\n";

  vector<string> sa { "Tokyo", "Osaka", "Fukuoka", "Nagoya" };
  string city {"Fukuoka"};

  size_t n1 { find(sa, city) };
  size_t n2 { find(sa, string("Nagoya")) };
  size_t n3 { find<string>(sa, "Osaka") };
  // size_t n4 { find(sa, "Nagoya") }; // これはエラー

  cout << n1 << " " << n2 << "" << n3 << "\n";
}
```

```
% ./a.out
3
2 3 1
```

# 関数テンプレート

- `int` 型と `string` クラスの双方用の `find()` 関数を、関数テンプレートによって一つにまとめて記述
  - ※定義が同じ関数をオーバーロードしなくて済む
- `T` がテンプレート引数であり、変数名と同様に任意の名前を付けられる
- 関数呼び出しのところで指定された実引数の型から、`T` に対応する型がコンパイラによって推測

```
1   size_t find(const vector<int>& a, int x) {
2     for (size_t i = 0; i < a.size(); i++)
3       if (a[i] == x) return i;
4     return a.size();
5   }
6
7   size_t find(const vector<string>& a, string x) {
8     for (size_t i = 0; i < a.size(); i++)
9       if (a[i] == x) return i;
10    return a.size();
11  }
```

↓ まとめる

```
1   template <typename T>
2   size_t find(const vector<T>& a, T x) {
3     for (size_t i = 0; i < a.size(); i++)
4       if (a[i] == x) return i;
5     return a.size();
6   }
```

# 関数テンプレートの使用

- 関数呼び出し `find(ia, 8)` から `T` を `int` として関数が生成
- 関数呼び出し `find(sa, city)` から `T` を `string` として関数が生成
- `"Nagoya"` の型は `char` 型の配列 → `string("Nagoya")` により、この文字列から `string` 型の値を作っている。

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

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

# 関数テンプレート：いつ使う？

- **関数テンプレートは多重定義された関数をまとめるテクニック**
- 関数テンプレート使わない（左）：同じ処理の関数が複数存在．実装が面倒．冗長．
- 関数テンプレート使う（右）：プログラマーは最低限の実装だけ．コンパイラが各関数を生成してくれる．

```
1   void print_add( int a, int b ) {
2     std::cout << a + b << "\n";
3   }
4
5   void print_add( double a, double b ) {
6     std::cout << a + b << "\n";
7   }
8
9   void print_add( string a, string b ) {
10    std::cout << a + b << "\n";
11  }
12
13  int main() {
14    print_add( 1, 2 );
15    print_add( 1.1, 2.2 );
16    print_add( "c++", "programming" );
17  }
```

```
1   template <typename T>
2   void print_add( T a, T b ) {
3     std::cout << a + b << "\n";
4   }
5
6   int main() {
7     print_add( 1, 2 );
8     print_add( 1.1, 2.2 );
9     print_add( "c++", "programming" );
10  }
```

※ 引数の型が違うだけで同じ処理をしている，と思った時には，関数テンプレートでまとめることを検討してみましょう．

# 最小値を見つける関数テンプレート

- テンプレート引数 `T` は型名の指定できる箇所ならどこでも使用可能
- 局所変数と戻り値の型にもテンプレート引数 `T` を指定
- `T()` は型 `T` のデフォルト値
  - `T` が `int` なら `0`

```
% ./a.out
2, empty case: 0
1.5, empty case: 0
BTW, empty case
```

```
1  template<class T>
2  T find_min(const vector<T>& a) { // 関数の戻り値が T型
3    if( a.empty() ) return T(); // T型のデフォルト値を返す
4
5    T min { a[0] }; // T型の変数を宣言
6    for( size_t i = 1; i < a.size(); i++ )
7      if( a[i] < min ) min = a[i];
8
9    return min;
10 }
11
12 int main() {
13   std::vector<int> x {3,5,2,4,7}, x_emp {};
14   std::cout << find_min(x) << ", empty case: " << find_min( x_emp ) <<"\n" ;
15
16   std::vector<double> y {8.2, 3.4, 1.5, 6.3, 7.5}, y_emp {};
17   std::cout << find_min(y) << ", empty case: " << find_min( y_emp ) <<"\n";
18
19   std::vector<std::string> z {"FYI", "EOM", "NRR", "PFA", "BTW"}, z_emp {};
20   std::cout << find_min(z) << ", empty case: " << find_min( z_emp ) <<"\n";
21 }
```

# 便利な関数テンプレート

- `<algorithm>` ヘッダファイルに，よく使う関数テンプレートが実装済み

`<algorithm>` に実装されているもの

```
// aとbの小さい方を返す
template<class T>
const T& min(const T& a, const T& b);

// aとbの大きい方を返す
template<class T>
const T& max(const T& a, const T& b);

// aとbを入れ換える
template<class T>
void swap(T& a, T& b);
```

使い方

```
1  #include <iostream>
2  #include <algorithm>
3
4  int main() {
5    double x{3.1}, y{4.2};
6
7    std::cout <<"min:"<< std::min(x, y) << ", "
8              <<"max:"<< std::max(x, y) <<"\n";
9
10   std::swap(x, y);
11   std::cout <<"x = "<< x <<", y = "<< y <<"\n";
12 }
```

# stringリテラル（sリテラル）

- 文字列リテラルと`string`の混乱をスマートに回避
- 接尾辞（サフィックス）`s`
- 文字列リテラルと`string`の関係に注意
  - 文字列リテラルは`char`型の配列
  - `string`は構造体の一種，文字列リテラルとは異なる型
- コンパイルには`-std=c++17`が必要

```cpp
// p11のプログラム
template <typename T>
size_t find(const vector<T>& a, T x) { /*省略*/ }

int main() {
  vector<string> sa { "Tokyo", "Osaka", "Fukuoka", "Nagoya" };
  string city {"Fukuoka"};

  size_t n1 { find(sa, city) };
  size_t n2 { find(sa, string("Nagoya")) };
  size_t n3 { find<string>(sa, "Osaka") };
  // size_t n4 { find(sa, "Nagoya") }; // これはエラー
  cout << n1 << " " << n2 << "" << n3 << "\n";
}
```

```cpp
// このように改善可能
using namespace std::string_literals;
template <typename T>
size_t find(const vector<T>& a, T x) { /*省略*/ }

int main() {
  vector sa {"Tokyo"s, "Osaka"s, "Fukuoka"s, "Nagoya"s};
  size_t n2 = find(sa, "Nagoya"s);
  cout << n2 <<"\n";
}
```

# 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**
- 見つからなかった場合には、コンパイラはすぐにエラーとはせず、以下のような型の変換などを試みて一致する関数がないかを調べる。
  - `bool, char` → `int`
  - `float` → `double`
  - `int` ↔ `double`
  - `int` → `unsigned int`

```cpp
void print( int x ) {
  std::cout << "int: " << x << "\n";
}
void print( double x ) {
  std::cout << "double: " << x << "\n";
}
void print( std::string x ) {
  std::cout << "str: " << x << "\n";
}

int main() {
  print( 10 );    // print( int x ) が呼ばれる
  print( 3.14 );  // print( doulble x ) が呼ばれる
  print( "cat" ); // print( string x ) が呼ばれる

  print( true );  // print( int x ) が呼ばれる
  print( 'a' );   // print( int x ) が呼ばれる
  float pi = 3.14;
  print( pi );    // print( double x ) が呼ばれる
}
```

# 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。