C++の関数テンプレート

開発していると、次のように操作対象のデータ型以外は同じように見えるコードが多数あることに気づくかもしれません。

```
#include <iostream>
#include <string>

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

double addDoubles(double a, double b) {
    return a + b;
}

std::string concatenateStrings(std::string a, std::string b) {
    return a + b;
}

int main() {
    std::cout << "Adding ints: " << addInts(3, 5) << '\n';
    std::cout << "Adding doubles: " << addDoubles(3.5, 1.2) << '\n';
    std::cout << "Adding strings: " << concatenateStrings("Hello",
"World") << '\n';

    return 0;
}
```

ここで定義されている各関数のロジックはまったく同じです。各関数の違いは、加算または結合するデータ型だけです。

このようなコードの重複はDRY原則に反しているため、コードをクリーンアップする方法が必要です。

C++の関数テンプレートを使用すると、さまざまなデータ型で機能する関数を作成できます。さまざまなデータ型で使用できる単一の関数を定義できるので、データ型ごとに異なる関数を書く必要はありません。

関数テンプレートを使用するには、最初にクリーンアップ対象となる全関数の共通機能を含んだ新しい関数を作成し、適切な名前を付けます。次に、関数シグネチャ内で変化する型をすべて削除し、それらをプレースホルダ型に置き換えます。

前のコードに対してこれを実行すると、次のようになります。

```
T add(T a, T b) {
    return a + b;
}
```

「add」は対象となるすべての関数の処理内容を表した適切な名前です。各関数で戻り値の型と2つの引数の型は変化しますが、関数内の戻り値と引数の型はすべて同じであるため、これらすべてにプレースホルダ型Tを使用しています。

このままコードをコンパイルしようとすると、Tを宣言していないためコンパイラエラーが発生します。

次の手順では、テンプレートヘッドを使用してTを宣言します。

```
template <typename T>
T add(T a, T b) {
    return a + b;
}
```

templateキーワードを使用してテンプレートヘッドを宣言し、その後<>の中に1つ以上のパラメータを書きます。この例のTは、データ型を表すテンプレート型パラメータです。ここでは、Tが型であることを示すためにtypenameキーワードを使用しています。非型テンプレートパラメータやテンプレートテンプレートパラメータもあります。

最後の手順では、関数を呼び出すときのテンプレートパラメータの型を指定します。

```
int main() {
    std::cout << "Adding ints: " << add<int>(4, 5) << '\n';
    std::cout << "Adding doubles: " << add<double>(3.7, 1.2) << '\n';
    std::cout << "Adding strings: " << add<std::string>("Hello", "World")
<< '\n';

    return 0;
}
```

前のコードの出力は次のようになります。

```
Adding ints: 9
Adding doubles: 4.9
Adding strings: HelloWorld
```

この出力から、作成した1つの関数をさまざまな型で使用できることがわかります。コンパイラは関数の呼び出し時に指定したデータ型に基づいて、関数の特定のバージョンを生成します。

では、渡された2つの値の積を返す関数テンプレート「multiply」を定義し、intとdoubleの両方の引数でこの関数を呼び出しましょう。

```
template <typename T>
T multiply(T a, T b) {
    return a * b;
}
```

```
int main() {
    std::cout << "Multiplying ints: " << multiply<int>(5, 6) << '\n';
    std::cout << "Multiplying doubles: " << multiply<double>(3.9, 1.3) <<
'\n';
}
```

テンプレートの実引数推定

コンパイラは *テンプレートの実引数推定* の仕組みを利用して(利用可能な場合)、テンプレートに渡されたデータ型を推定できます。このため、一般的には通常の関数と同じように関数テンプレートを呼び出すことができます。

```
template <typename T>
void log(T data) {
    std::cout << "Logging data: " << data << '\n';
}

int main() {
    int int_data = 3;
    log(int_data);

    double double_data = 3.14;
    log(double_data);
}
```

```
template <typename T, typename U>
auto max(T x, U y) {
    return (x > y) ? x : y;
}

int main() {
    int int_a = 3;
    int int_b = 5;
    std::cout << max(int_a, int_b) << std::endl;

    int double_a = 3.14;
    int double_b = 5.25;
    std::cout << max(double_a, double_b) << std::endl;
}
```

関数テンプレートの特殊化

add関数テンプレートは単に2つの文字列を結合します。これは「+」演算子の動作と同じです。しかし、2つの単語の間にスペースを入れたい場合はどうでしょうか。

これを実現するには、add関数の *テンプレートの特殊化* を定義します。特殊化では、テンプレートパラメータ宣言を空のままにして、型を明示的に指定します。

```
template <>
std::string add(std::string a, std::string b) {
    return a + " " + b;
}
```

コンパイルして前のコードを再度実行します。

```
int main() {
    std::cout << "Adding ints: " << add<int>(4, 5) << '\n';
    std::cout << "Adding doubles: " << add<double>(3.7, 1.2) << '\n';
    std::cout << "Adding strings: " << add<std::string>("Hello", "World")
<< '\n';

    return 0;
}
```

次のような結果になります。

```
Adding ints: 9
Adding doubles: 4.9
Adding strings: Hello World
```

演習

演習1

次の関数テンプレートについて考えてみましょう。

```
template <typename T>
T add(T a, T b) {
    return a + b;
}
```

2つの浮動小数点引数を使用してaddを呼び出す正しい方法は次のどれですか。

```
a) add<float>(3.0, 4.0)
b) add(3.0f, 4.0f)
c) add<float, float>(3.0, 4.0)
d) add(3.0, 4.0f)
```

演習2

同じ型の2つの引数を受け取り、小さいほうの値を返すC++の関数テンプレート「min」を書いてください。

演習3

同じ型の2つの変数の値を入れ替える関数テンプレート「swap」を書いてください。

演習4

次のコードがあります。

```
#include <iostream>
#include <vector>

template <typename T>
T sum(const std::vector<T> &data) {
    T result = 0;
    for (const auto &item : data) {
        result += item;
    }
    return result;
}

int main() {
    std::vector<int> intVec = {1, 2, 3, 4, 5};
    std::vector<double> doubleVec = {1.1, 2.2, 3.3, 4.4, 5.5};
    std::vector<std::string> stringVec = {"one", "two", "three"};

    std::cout << "Sum of intVec: " << sum(intVec) << '\n';
    std::cout << "Sum of doubleVec: " << sum(doubleVec) << '\n';
    std::cout << "Sum of stringVec: " << sum(stringVec) << '\n';

    return 0;
}
```

このコードは、関数テンプレートを使用してvector内の要素の合計を計算します。コードはそのままコンパイルされますが、コードの実行時にセグメンテーション違反が発生します。問題を特定して修正し、正常に動作するコードにしてください。

演習5

次のコードを評価し、関数テンプレートの適切なユースケースかどうか説明してください。適切でない場合は、より適切な代替案を提案してください。

```
#include <iostream>

template <typename T>
void print(T data) {
    std::cout << data << std::endl;
}
```

```
int main() {
    print<int>(42);
    print<double>(3.14);
    print<std::string>("Hello, World!");

    return 0;
}
```