

第1回C++講座

- C++はオブジェクト指向言語と言われている
- 実際には
C言語的な記述
オブジェクト指向プログラミング
関数型プログラミング
テンプレート・メタ・プログラミング
など、多彩な表現技法が使える言語

よく言えば柔軟で、悪く言えば統一感がない
この講座では上二つを対象にする

1.C言語とC++の違い

```
#include <cstdio>
int main()
{
    int x, y;
    puts("Hello world!!!);

    double f = 33;
    return 0;
}
```

```
#include <stdio.h>
void test() {}
int main()
{
    test(1);
}
```

- (1) 標準ヘッダに ".h" をつけない
 - (2) C言語の標準ヘッダは、もとの名前の先頭に 'c' をつける (例) <cstring>, <ctime>
※<stdio.h> のようにしても問題ない
 - (3) 引数の void は省略できる
 - (4) スコープの途中で変数が宣言できる
 - (5) 構造体変数の宣言にstruct が必要ない
 - (6) bool 型と true, false
 - (7) 便利な標準ライブラリが存在する
(検索キーワード: STL)
- (問) 左のプログラムは、C言語とC++でどのような違いがあるか

関数の多重定義

```
#include <stdio>
void function() {puts("void"); }
void function(double) {
    puts("double");
}
void function(char*) {puts("char*");}

int main() {
    function();
    function(1);
    function(1.0);
    function("aiueo");
    return 0;
}
```

• ポイント

- (1) 同じ名前で、引数が違う関数を複数定義できる
- (2) どの関数を使用するかは自動で判別される。型の一致する関数が優先的に選ばれ、なければ暗黙の型変換をして関数を選ぶ。
- (3) 暗黙の型変換等で、どの関数を使用するか曖昧な場合はコンパイルエラーが起こる
- (4) 引数が同じで返り値が違う関数は定義できない
- (5) つまり、どの関数を使用するか判別できることが大事

※暗黙の型変換 が同じ引数に対して二度行なわれることはない(クラスで詳しく)

演習

```
#include <stdio>
void function() {};
int function() {return 0;}

int main()
{
    function();
    return 0;
}
```

1. 左のプログラムはどのような問題があるか
2. int 型、double 型、文字列の交換関数 swap を作成せよ
(ヒント: strcpy を使用してよい)
3. 名前、ID、年齢、身長、体重をメンバに持つ Person 構造体を定義し、swap 関数を作成せよ
4. 引数として二つの long 型を受け取り、大きい方の値を返す maxof 関数を作成せよ。同様の関数を double 型について作成し、整数リテラルについて動作を確認せよ。キャストしたらどうなるか
5. 三角形の面積を求める area 関数の宣言を、
 - (a) 三辺の長さから求める場合
 - (b) 正三角形の場合について記述せよ

名前空間

```
// quickLib.h
namespace quickLib
{
    int x;
    void print();
    void read();
    void write();
}
```

```
// secureLib.h
namespace secureLib
{
    int x;
    void print();
    void read();
    void write();
}
```

• ポイント

- (1) 同じ名前の変数、同じ引数の関数を多重定義できる
- (2) つまり、複数のライブラリをインクルードしても競合しない
- (3) 使用するときには、“::” を用いて以下のように書く

```
int main()
{
    quickLib::print();           // quickLib の print()
    secureLib::print();         // secureLib の print()
    return 0;
}
```

- (4) アドレス解決演算子

“::” は、どの名前空間の要素かを指定する(クラスでも使う)

- (5) main関数のように名前空間を使用していない場合、グローバルな名前空間に存在しているといい、全ての名前空間から参照される

- (6) 以下の場合はアドレス解決演算子が不要。

- (a) 自分の所属する名前空間の要素
- (b) グローバルな名前空間の要素
- (c) using で指定した名前空間の要素
にアクセスするとき。

コンソール入出力

```
#include <iostream>
int main()
{
    int x;
    std::cout << “数値を入力してください : “;
    std::cin >> x;
    std::cout << “入力されたのは “ << x << “です\n”;
    return 0;
}
```

```
#include <iostream>
using namespace std;
int main()
{
    int x;
    cout << “数値を入力してください : “;
    cin >> x;
    cout << 入力されたのは “ << x << “です\n”;
    return 0;
}
```

• ポイント

(1)左のプログラムは全く同じ内容

(2)<iostream> ヘッダ

C言語の<stdio.h> に相当する

(3) cout, cin

標準入出力

cout, cin は、std という名前空間に定義されている

(4) using namespace std;

は、std 名前空間にある全要素に、
:: なしでアクセスするためのもの

(5) 入出力する変数の型は

cin, cout により自動的に判別される

※using namespace の使用 は、名前空間の利点である名前の衝突の回避を無効化してしまう。

新しいキャスト

```
#include <iostream>
using std::cout;
// std のうち、cout のみ”::” なしで使用

int main()
{
    int x = 100;
    const int* pp = &x;
    int* p = reinterpret_cast<int*>(x);
    double f = static_cast<double>(x);
    p = const_cast<int*>(pp);
    return 0;
}
```

- `static_cast`
互換性のある型への変換
- `reinterpret_cast`
互換性のない型への変換
- `const_cast`
const型から非const 型への変換
- `dynamic_cast`
基底クラス型から派生クラス型への変換

ポイント

- (1) `xx_cast<型名>(変数名);`
- (2) それぞれの変換を安全に行なう
- (3) 見ただ目で意図がはっきりする
- (4) Cスタイルのキャストでも変換はできる

演習

- 名前空間myStd に、main という名前の関数を作成せよ。
コンパイルし、その結果の意味を考察せよ
- 名前空間 myStd に、power関数を作成せよ。関数はdouble 型の変数xと、int 型の変数 yを受け取り、xのy乗を返す。(y >= 0)
- 二つのdouble 型変数をメンバに持つPoint構造体を作成し、これを用いて二点の距離を返す関数を作成せよ。
- 三つのPoint構造体をメンバに持つ Triangle構造体を作成し、その面積を返す関数を作成せよ。

※C++標準入出力を使用すること

ヒント:ヘロンの公式

三辺の長さをa, b, c とする。

$2s = a + b + c$ 、面積をAとすれば、

$$A = (s * (s - a) * (s - b) * (s - c))^{(1 / 2)}$$

オブジェクトとインスタンス

```
#include <iostream>
struct BigData
{
    char bigBit[10000];
};

int main()
{
    BigData source;
    for(int i = 0; i < 10000000; ++i)
    {
        BigData temp = source;
    }
    return 0;
}
```

- クラス定義、構造体定義をそれぞれオブジェクトと呼ぶ。データの集まり、処理の集まりごとにプログラムを纏めていくのがオブジェクト指向？
- インスタンスとは、オブジェクトの変数のことで、実際にメモリ上に確保されたオブジェクトのことを指す
- 値渡し、代入、初期化では、メンバのビットコピーが行なわれる
- 配列も問題なくコピーされるが、大きいデータのコピーは時間がかかる

構造体を引数に取る場合の注意

```
#include <iostream>
typedef struct {
    int * pointer;
} TestData;
void test(TestData t) {
    *(t.pointer) = 0;
}
int main() {
    int a = 10;
    TestData data = {&a};
    test(data);
    std::cout << data.pointer;
    return 0;
}
```

• ポイント

- (1) ポインタ渡しは、ポインタ変数の値がコピーされている。
- (2) ポインタ渡しでなくても、メンバにポインタを持つ構造体を渡す場合は、そのポインタを経由して値が書き換えられる可能性がある
 - * 構造体のメンバ自体は変化していない
- (3) ポインタ渡しを使う状況は、
 - (a) 変数の値を変更するとき
 - (b) 配列を用いるとき
 - (c) 値渡しでは処理速度に問題があるとき(一般に参照を用いる)
- (4) ポインタ渡しでは、値を変更しない場合は常に `const` を用いると良い

参照

```
#include <iostream>
void swap(int &a, int &b) {
    temp = a;
    a = b;
    b = temp;
}

int main() {
    int x = 3, y = 4;
    int &ref = x;
    ref = 10;
    swap(x, y);
    std::cout >> x >> ' ' >> y >> '\n';
    return 0;
}
```

• ポイント

- (1) クラス名 &変数名 と宣言する
- (2)参照は、変数に別の名前をつけることに相当する
- (3)参照は、常に*をつけたポインタと同じである。
- (4)参照変数は、普通の変数と同じように使用する。
- (5)参照は初期化しなければならない
- (6)同時に、参照にはNULLが存在しない
- (7)ポインタ同様、参照している変数の生存期間に注意する
- (8)参照渡しは変数をコピーしない

※左の例では、ref の値を変更するとaの値も変わる

演習

```
int *getIntPointer() {
    int temp = 100;
    return &temp;
}
int &getIntReference() {
    int temp = 100;
    return temp;
}
int getInt() {
    int temp = 100;
    return temp;
}
```

1. 左のプログラムの間違いを指摘せよ。また、どのような問題が起こるか予想し、実際に試してみよ
2. コンストラクタを学んでから、参照の動作について試してみよ
3. 別紙の問題を解き、配列に関する復習をする
4. 参照とポインタの違いについて考察せよ

宣言と定義

• クラス、関数の宣言

何度でも宣言できる。

クラスの宣言は、

(1)そのクラスのポインタの使用

(2)関数の引数として使用

する場合に必要

関数の宣言は、関数を使用

する場所より後ろで関数を定義

する場合に必要

※変数の宣言は、一度だけ。

• クラス、関数の定義

定義は一度だけ。二度目以降は
コンパイルエラーになる。

クラスの定義は、

(1)変数を宣言するとき

(2)配列を宣言するとき

に必要

関数の定義は、

プログラムをリンクするときに必要

※プログラムは、コンパイル→リンク

という過程を経て .exe になる

クラスの基礎

- クラスとは、プログラムの処理に合わせて作ることができるユーザー定義型
- C言語での構造体のように、任意の型の変数を保持するデータ型として定義できる。保持する変数をメンバ変数という
- さらに、それらのデータ型をどのように操作するかも定義できる。クラスに定義された関数をメンバ関数と呼ぶ
- クラス定義はあくまで型の定義で、定義した時点ではメモリ上には何も存在しない

クラス定義の書式

```
class クラス名
{
  アクセス修飾子 :
    メンバ変数;
    メンバ関数;
};
```

* C++では、クラスと構造体は同じもの！

- ポイント
 - (1) クラス名、メンバ変数名、メンバ関数名は任意で、クラス名が型名になる
 - (2) 要素数、記述の順番は任意
 - (3) アクセス修飾子の後ろはコロン
 - (4) 構造体同様、定義の最後にはセミコロン
 - (5) 変数として使用できるのは、定義が存在する型のみ
 - (6) 同一クラスの保持は、定義が終わっていないので出来ない。出来たとしても無限再帰になる
 - (7) 同一クラスのポインタは保持できる(サイズが決まっている、型が違う、ポインタは宣言だけで使用できるから)

メンバ関数とは

- メンバ関数とは、クラスのメンバ変数进行操作するための関数である
- メンバ変数を直接操作するのがC言語での構造体。同じ事もできるし、メンバ変数の操作を制限するためにメンバ関数を使用することもできる
- 構造的には、メンバ関数は、自身を引数に取る関数と同じである(例を参照)
- 別のクラス、別の名前空間の関数と名前の衝突を起こさない
- メンバ変数はそれぞれのインスタンスごとに存在し、メンバ関数はそのメンバ変数に対して処理をする

アクセス修飾子

- **private**

メンバ関数からのみアクセス可能

- **protected**

メンバ関数、派生クラスのメンバ関数からのみアクセス可能

- **public**

どこからでもアクセス可能

- 何も指定しないとき

- (1) クラス

指定されていない要素は全てprivate

- (2) 構造体

指定されていない要素は全てpublic

アクセス修飾子の意義

- インターフェースと詳細を分ける意味合いがある
- 中のデータには直接触れて欲しくない場合に、メンバ関数からのアクセスのみに制限できる(カプセル化)
- あとで直接いじりたくなるかもしれない。全部 public にしておいて、使うとき注意していればいいのか？
 - 実装の変更がきかなくなる。注意してても忘れる
- どのメンバにどのアクセス修飾子を使うかは状況による

演習

- int 型の変数を100個格納できる queue クラスを作成しなさい。
- push 関数…キューに値を入れる。成功したら true を返し、キューがいっぱいなら false を返す。
- pop 関数…キューから値を取り出す。成功したら true を返し、キューが空なら false を返す。
。

演習

1. 生徒を表わす Student クラスを作成せよ。名前、ID番号、学年と、このデータが有効かどうかを表わすフラグをメンバ変数として持ち、ID番号が負数の場合にはそのデータは無効であるとする。

標準入力を用いて、このクラスの各要素に代入を行ない、標準出力を用いて結果を表示せよ。データが無効である場合はその旨を表示すること。

2. 前問の Student クラスについて、データの有効性を表わすフラグが必要でないと気付いた。

このクラスからデータの有効性を表わすフラグを削除せよ。また、どのようなクラス構成であれば、他のプログラムの改変を最小限に出来たか考えてみよう。

3. Timer クラスを作成せよ。計測を開始する `start()`、計測を終了する `end()`、計測結果を返す `getResult()`、経過時間を返す `getPastTime()` の四つの関数をメンバ関数として定義すること。

この際、外部からメンバ変数にアクセスする必要があるか考えよ

* `ctime` ヘッダの `time(0)` 関数を使用する

解答

```
namespace myStd {  
    int main() { return 0;}  
}
```

コンパイルエラー: グローバル名前空間に
main関数がない(mainの代わりとして)
コンパイル可能: mainとは名前空間が違う }

```
namespace myStd {  
    double power(double x, int y) {  
        if(y = 0) return 1.0;  
        for(int i=0; i < y; ++i) {  
            x *= x;  
        }  
        return x;  
    }  
}
```

```
struct Point {  
    double x; double y;  
};  
double distance(struct Point p1, struct Point p2) {  
    return sqrt(myStd::power(p1.x - p2.x, 2) +  
        myStd::power(p1.y - p2.y, 2));  
}
```

```
struct Triangle {  
    Point p1; Point p2; Point p3;  
};  
double area(Triangle t) {  
    double a = distance(t.p1, t.p2), b =  
        distance(t.p2, t.p3), c = distance(t.p3, t.p1);  
    double s = (a + b + c) / 2;  
    return sqrt(s * (s - a) * (s - b) * (s - c));  
}
```

解答

Cでは実行可能。引数の省略は、
int 型引数を1つ取るものまたはvoid と
解釈される

C++ではコンパイルエラー。引数の省略
は、void と解釈される

戻り値の型が異なるだけでは
多重定義できないのでコンパイルエラー

```
#include <cstring>
void swap(int* a, int *b) {
    int temp = *a;
    *a = *b;
    *b = temp;
}
```

```
void swap(char str1[], char str2[]) {
    char temp[128];
    strcpy(temp, str1);
    strcpy(str1, str2);
    strcpy(str2, temp);
}
```

解答

```
struct Person {
    char name[128];
    int age;
    double height;
    double weight;
};

void swap(struct Person *a, struct Person *b)
{
    Person temp; // struct は省略可能
    temp = *a;   // 代入はビットコピー
    *a = *b;
    *b = *a;
}
```

```
int maxof(double a, double b) {
    puts("double maxof");
    return a > b ? a : b;
}

long maxof(long a, long b) {
    puts("long maxof");
    return a > b ? a : b;
}

int main()
{
    maxof(1, 3);      // コンパイルエラー
    maxof((double)1, (double)3);
    maxof((long)1, (long)3);
    return 0;
}

double area( double len1, double len2, double len3);
double area(double length);
```