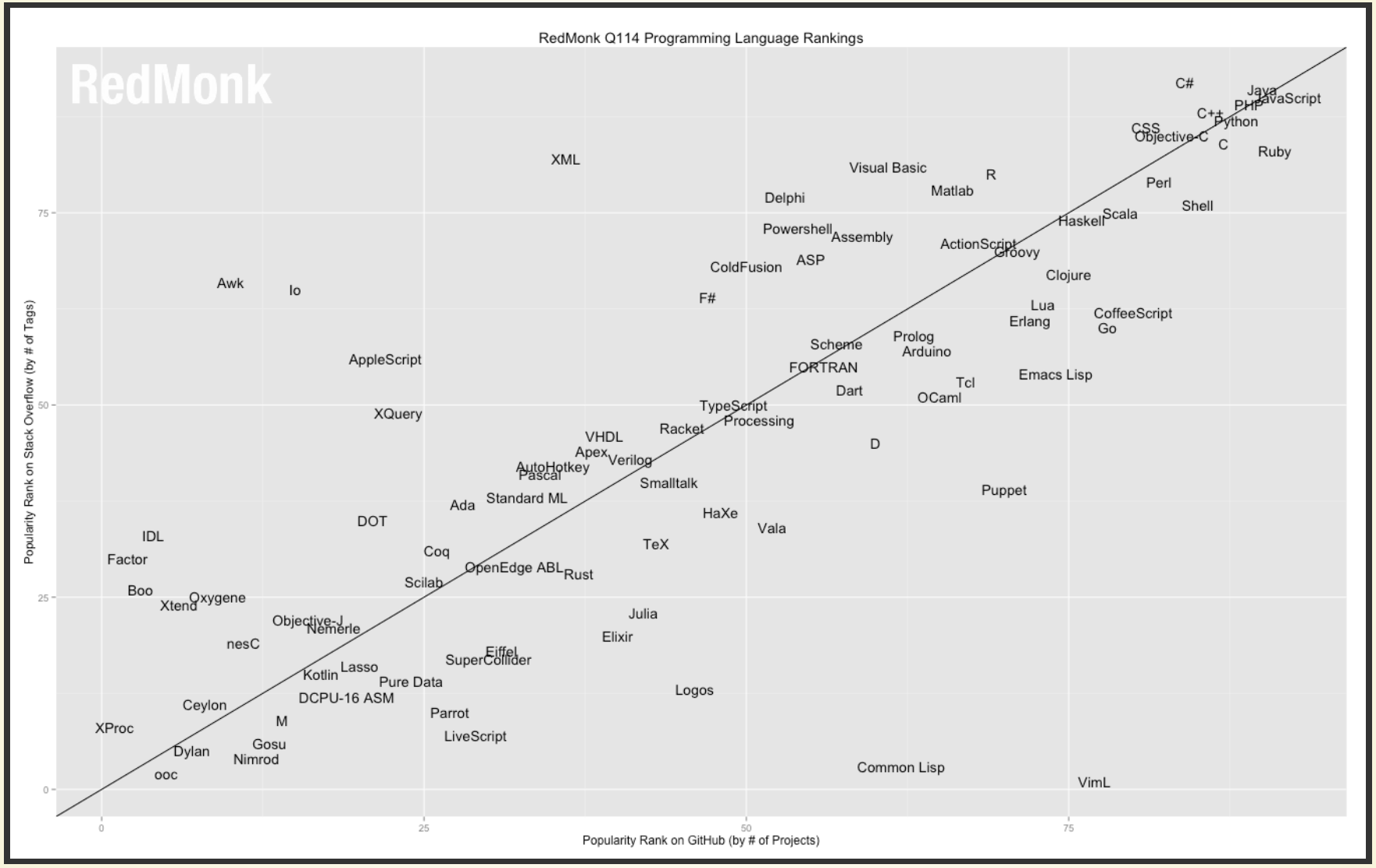
C++プログラミング基礎

2014/4/28

目次

- 型、クラス、データ構造
  - プリミティブ型
  - クラス
  - データ構造
- スコープ
- オブジェクト指向プログラミング
  - カプセル化
  - 継承
  - ポリモーフィズム
- プログラムが実行されるまで

どんな言語がある？


RedMonk Q114 Programming Language Rankings
RedMonk
Popularity Rank on Stack Overflow (by # of Tags)
Popularity Rank on GitHub (by # of Projects)
0
25
50
75
C#
Java
JavaScript
PHP
C++
Python
CSS
Objective-C
C
Ruby
XML
Visual Basic
R
Perl
Matlab
Delphi
Shell
Haskell
Scala
Powershell
Assembly
ActionScript
Groovy
ASP
ColdFusion
Clojure
Awk
Io
F#
Lua
Erlang
CoffeeScript
Go
Scheme
Prolog
Arduino
AppleScript
FORTRAN
Emacs Lisp
Tcl
Dart
OCaml
XQuery
TypeScript
Processing
Racket
VHDL
Apex
Verilog
D
AutoHotkey
Pascal
Smalltalk
Puppet
Ada
Standard ML
HaXe
DOT
Vala
IDL
TeX
Coq
Factor
OpenEdge ABL
Rust
Scilab
Boo
Oxygene
Xtend
Julia
Objective-J
Nemerle
Elixir
nesC
Eiffel
SuperCollider
Kotlin
Lasso
Pure Data
Logos
DCPU-16 ASM
Ceylon
Parrot
XProc
M
LiveScript
Gosu
Dylan
Nimrod
Common Lisp
ooc
VimL

C++言語

Bjarne Stroustrup　が1983年にC言語を拡張して開発

- それなりに人気
- 実行速度が速い
- 言語仕様が進化し続けている，古くて新しい言語
  - C++98，C++03，C++TR1，C++11，C++14...

今後の説明のための基礎知識

- コメント

```
コメントじゃない //行末までのコメント          コメント

コメントじゃない /*囲まれた範囲内のコメント*/ コメントじゃない
```

- 関数

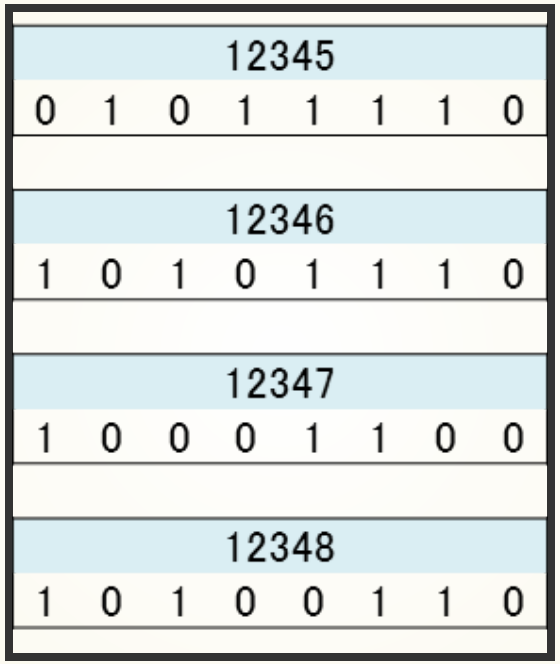
```
int func(int a) {
    int b = 3;
    return a * b;
}
```

- 標準出力

```
#include <iostream>

int main() {
    std::cout << "Hello World" << std::endl;
    //-> Hello World
}
```

メモリとアドレス

プログラムで用いる値はコンピュータ上のメモリに保管される．メモリには8bit(1byte)ごとにアドレスが割り振られている



型、クラス、データ構造

型とオブジェクト

プログラムを構成する要素(オブジェクト)は型に基づいて分類される

C++ではオブジェクトの型は変更されない(静的型付け)

- プリミティブ型

```
char c = 'c';  //文字型
int i = 1;  //整数型
float f = 3.1415;  //浮動小数点型
double d = 3.1415;  //倍精度浮動小数点型
```

- コンテナ型(特別なクラス、STLにより実現)

```
std::string s("hello world");  //文字列型
std::vector v(1, 2, 3);  //可変長配列
std::list l(1, 2, 3);  //双方向リスト
//  e.t.c.
```

※関数もオブジェクト(特別なクラスにより実現)

```
auto plus_one = [] (int x) -> int {return x + 1};
```

プリミティブ型 (組み込み型)

言語に予め備わっている基本的な型

- void
- signed char, unsigned char
- short, unsigned short, int, unsigned int, long long, unsigned long long
- float, double

値の初期化

```
int a = 1;  //数値で初期化
char b = 'b';  //文字で初期化
int arr0[3] = {1, 2, 3};  //初期化子リストで初期化
std::string c = "Hello world";  //文字列リテラルで初期化
```

明示的に初期化しなければ値は未定義の闇へ

```
int a;
std::cout << a ;  //0であるとは限らない
```

(プリミティブ型の)オブジェクトは使う前に必ず初期化しよう！

クラス

データとそれに対する操作をセットにした型

- データ:メンバ変数
- 操作:メンバ関数

```
class Hito {
    private:
        std::string name;//名前
        int age;//年齢
    private:
        Hito();//コンストラクタ
        ~Hito();//デストラクタ
    public:
        void setname(const std::string &str);//名前の設定
        void disp();//名前と年齢の表示
}
```

上記のコードをクラスの宣言という

クラスの実装

メンバ関数の処理を記述する

```
//名前の設定
void Hito::setname(std::string str) {
    name = str;
}

//名前と年齢の表示
void Hito::disp() {
    std::cout << "name:" << name << ", " << "age:" << age << std::endl;
}
```

Hito::でHitoクラスのメンバ関数を実装していることを明示して、それ以外は普通の関数と同じように記述

宣言と同時の実装

- クラスの宣言と実装を分けずに同時に記述することができる
- 極小規模のプログラム，クラス内クラスなどで用いられる
- 翻訳単位の問題があるので通常は使わない．

```
class Hito {
    private:
        std::string name;
        int age;
    private:
        Hito() {/*コンストラクタの記述*/}
        ~Hito() {/*デストラクタの記述*/}
    public:
        void setname(std::string str) {
            name = str;
        }
        void disp() {
            std::cout << "name:" << name << ", " << "age:" << age << std::endl;
        }
}
```

クラスの使い方

```
int main (void) {
    Hito hito;      //クラスからオブジェクトを作ることを「インスタンス化」という
    // -> 山田太郎17歳で初期化しました

    hito.disp();
    // -> name:Taro Yamada, age:17

    hito.setname("Ichiro Suzuki");
    hito.disp();
    // -> name:Ichiro Suzuki, age:17
}
```

オブジェクトの配列にもできる

```
int main (void) {
    Hito hito_array[20];
    hito_array[0].setname("Hanako Yamada");
    hito_array[1].setname("Taro Yamada");

    hito_array[0].disp();
    hito_array[1].disp();
}
```

関数の引数にもできる。大体なんでもできる。

```
void func (Hito hito) {
    hito.setname("FUNCTION");
    hito.disp();
}
```

特殊なメンバ関数

- コンストラクタ
  - オブジェクトが生成されるときに実行されるメンバ関数
  - クラスと同じ名前の関数で表す

```
Class::Class();
```

- デストラクタ
  - オブジェクトが破棄されるときに実行されるメンバ関数
  - クラスの名前に「~」を付けた関数で表す

```
Class::~Class();
```

- 演算子オーバーロードされたメンバ関数
  - 演算子のように振舞うメンバ関数

```
Class::operator=();
Class::operator+();
e.t.c...
```

コンストラクタ

- オブジェクトが生成されるときに自動的に実行される
- 主にメンバ変数の初期化に用いられる

```
Hito::Hito() {
    name = "Taro Yamada";
    age = 17;
    std::cout << "山田太郎17歳で初期化しました" << std::endl;
}
```

メンバ変数の初期化は以下のようにも書ける(推奨)

```
Hito::Hito() : name("Taro Yamada"), age(17) {
    std::cout << "山田太郎17歳で初期化しました" << std::endl;
}
```

コンストラクタに引数を与えることもできる

```
Hito::Hito(std::string name, int age) : name(name), age(age) {
    std::cout << "山田太郎17歳で初期化しました" << std::endl;
}
```

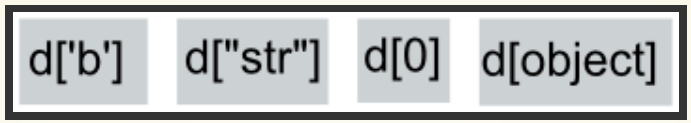
データ構造
可変長配列(VECTOR)

- プログラムの実行中に長さを変えることができる配列
  - あるインデックスのデータへのアクセスは速い
  - 要素の挿入、入れ替えには時間がかかる

```
//必要なヘッダファイル
#include<vector>
```

```
//可変長配列の生成
std::vector<int> v;

//末尾に値を追加
v.push_back( 1 );
v.push_back( 2 );
v.push_back( 1 );

for (int i = 0; i < 3; ++i) {
    std::cout << v[i] << ", ";
}
// -> 1, 2, 1

std::cout << v.size();
```

連想配列



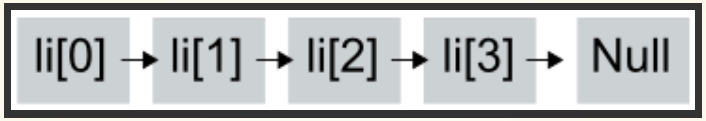
- キー(key)とデータを対応付けることができる配列
  - インデックスに数字以外を使える

```
//必要なヘッダファイル
#include<map>
```

```
//連想配列の生成
std::map<string, int> height;

//高さの登録
height["Fuji"] = 3776;
height["Takao"] = 599;
```

```
//高さの読み出し
std::cout << height["Fuji"] << std::endl;
// -> 3776
```

リスト



- 値に順序があるコンテナ
  - 要素の挿入、並べ替えが速い

```
//必要なヘッダファイル
#include<list>
```

```
//連結リストの生成
std::list<int> valuelist;

//要素の追加
valuelist.push_back( 7 );
valuelist.push_back( 5 );
valuelist.push_back( 6 );

//要素の並べ替え
valuelist.sort();

for (std::list<int> it = valuelist.begin(); it != valuelist.end(); ++i) {
    std::cout << *it << ", ";
}
// -> 5, 6, 7
```

スコープと変数の寿命

スコープ(名前空間)

プログラムの実行単位において見える範囲

```
int g;

int main (void) {

    for (int i = 0; i < 5; ++i) {    std::cout << i;    }

    {    int i = 1;    }

    int i = 10;
}
```

- 変数は使いたい位置で宣言できる
- 自動変数はスコープから抜けると破棄される。そのため、スコープが異なれば名前は衝突しない。

必要な変数は必要な場所で！ 適切なスコープを実現するように宣言の位置に気をつけることで、読みやすい、バグが少ないプログラムにできる！

メンバ変数、メンバ関数のスコープ

アクセス修飾子

```
private:    //クラスの中からしか見えないメンバ
public:     //クラスの外からも見えるメンバ
```

クラスの宣言時に指定する

```
class TEST {
    private:
        int privalue;       //プライベートなメンバ変数
        void prifunc();     //プライベートなメンバ関数
    public:
        int pubvalue;       //パブリックなメンバ変数
        void pubfunc();     //パブリックなメンバ関数
}
```

```
int main() {
    TEST test;
    int val = test.pubvalue;
    test.pubfunc();
    int val = test.privalue;    //コンパイルエラー
    test.prifunc();             //コンパイルエラー
}
```

この他のアクセス修飾子には、「protected」がある(後述)

名前のあるスコープ「名前空間」

スコープは異なる同じ名前の変数

```
{   int val = 10;  }

{   int val = 20;  }

std::cout << val << std::endl;    //コンパイルエラー ここからはアクセスできない
```

「名前空間」を用いてスコープに名前をつける

```
namespace A {   int val = 10;   }

namespace B {   int val = 20;   }

std::cout << A::val << std::endl;
// -> 10

std::cout << B::val << std::endl;
// -> 20
```

スコープ解決演算子「::」でそれぞれにアクセスできる

変数の寿命

- スコープを抜けると自動的に削除される変数を「自動変数」または「ローカル変数」という

```
{    int a;   }
```

- プログラムが終了するまで消去されない変数を「静的変数」という

```
int g;  //グローバル変数

int func() {
    static int sti;     //static変数
}
```

- グローバル変数や,「static」と付けられた変数は静的変数になる
- プログラムの見通しを悪くする傾向があるので必要かどうかよく考えるべき

静的変数

- 関数内の静的変数

```
void func() {
    static int i;
    std::cout << i << std::endl;
}

int main() {
    func();
    //-> 0;
    func();
    //-> 1;
}
```

- 静的メンバ変数

```
class Obj {
    static int i;
}

int main() {
    Obj a, b;
    std::cout << a.i << std::endl;
    //-> 0
    b.i = 8;
    std::cout << a.i << std::endl;
    //-> 8
}
```

寿命を自分で管理する変数

newしてからdeleteするまで使える

- メモリの動的確保 new/delete

```
int *p = new int();      //int型の領域を動的確保
*p = 1234;
std::cout << *p;

delete p;                //動的に確保した領域を解放
std::cout << *p;         //エラー
```

- 配列の場合

```
int *p = new int[10];
delete [] p;
```

- 動的なインスタンス化

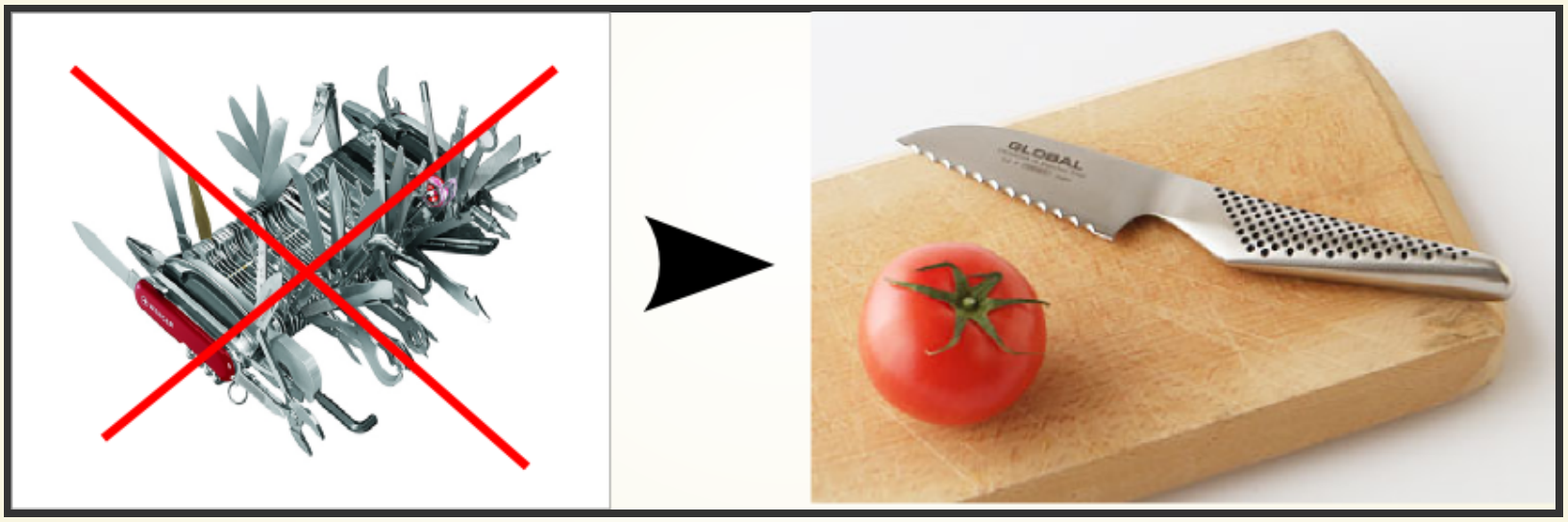
```
Object *p = new Object();
delete p;
```

※newした領域は忘れずdeleteしないとメモリリークを起こすプログラムになる！

オブジェクト指向プログラミング

オブジェクト指向とは

- C++のクラスのような概念のもとで、データとそれに対する操作をひとまとめにする方法
- プログラマがミスをしにくくする
- 文法というよりプログラムの書き方のテクニック



カプセル化

データを隠蔽して変更に強いプログラムにする方法

```
class Object {
    private:
        int cost1, cost2;    //隠蔽してクラス外から見えなくする
    public:
        int getsum() {  return cost1 + cost2;   }   //変数を見るにはこの関数を使う
}
```

プログラムの変更によってデータの数が変わってもデータを利用する側には影響がない

```
Object oyatsu;
int sum = oyatsu.getsum();
```

もしカプセル化されてなかったら...

```
class Object {
    public:
        int cost1, cost2, cost3;
}
```

データを利用する側のコードにすべて修正が必要になってしまう

```
Object oyatsu;
int sum = oyatsu.cost1 + oyatsu.cost2 + oyatsu.cost3;
```

継承

- クラス間の親子関係を作る方法
  - DRY(Don't repeat yourself)原則を守ってコードの重複を少なくする
  - プログラムの構造を見やすく、理解しやすくする

親クラス(スーパークラス)

```
class Animal {
    private:
        int weight;     //体重
        int height;     //身長
    public:
        int getweight();
        int getheight();
}
```

子クラス(サブクラス)

```
class Dog : public Animal {
    void walk() {   /*犬の歩き方*/  }
    void run() {    /*犬の走り方*/  };
}
```

使い方

```
Dog dog;
```

継承とコンストラクタ

親クラスのコンストラクタが子クラスの前に呼ばれる。

```
//親クラス
class Animal {
    Animal() {  std::cout << "Animal" << std::endl; }
}
//子クラス
class Dog : public Animal {
    Dog() {     std::cout << "Dog" << std::endl;    }
}

Dog dog;
// -> Animal
//    Dog
```

継承とスコープ

親クラスのメンバを子クラスのみから使えるようにするスコープ解決演算子「protected」

```
class Animal {
    protected:
        int weight;     //体重
}
class Dog : public Animal {
    int getweight() { return weight; }  //親クラスのメンバ変数が見える
}
```

アップキャスト

Animalクラスを継承したDogクラスとCatクラスがあるとする。

```
class Animal {
    private:
        int weight;     //体重
    public:
        int getweight();
}
```

Dog、Catの体重を表示する関数は以下のようになる。

```
void dogShowWeight(Dog dog) { std::cout << dog.getweight() << std::endl; }//犬用
void catShowWeight(Cat cat) { std::cout << cat.getweight() << std::endl; }//猫用
```

まとめられないか？アップキャストで解決↓ 親クラスの参照やポインタでキャストしてまとめて扱うことができる

```
void showWeight(Animal &animal) { std::cout << animal.getweight() << std::endl; }
```

犬だろうが猫だろうが動物として扱うということ

```
Dog dog;
Cat cat;
showWeight(dog);    //Dogのweightが表示される
showWeight(cat);    //Catのweightが表示される
```

オーバーライド

親で「仮想関数」になっているメンバ関数を子で書き換える

```
class Animal {
    public:
        virtual void walk() { /*動物の歩きかた*/ };
}
```

「virtual」と書くと仮想関数になる。「動物は歩くものだが、その歩き方は犬と猫では異なる」。子クラスごとに実装が違うべき共通の概念を表すときに仮想関数を用いる。

```
class Dog : public Animal {
    public:
        void walk() { /*犬の歩きかた*/ }
}
```

親クラスの仮想関数と同じ名前のメンバ関数を子クラスで宣言することを「オーバーライド」という。

```
Dog dog;
dog.walk(); //犬の歩き方が実行される
```

オーバーライドすると子クラスの処理が実行される。「virtual」がなければコンパイルエラーになる。

純粋仮想関数

- 実装のない関数
- サブクラスで必ず実装しなければならない

```
class Animal {
    public:
        virtual void walk() = 0;
}
```

「＝０」と書くことで純粋仮想関数になる

```
class Dog : public Animal {
    public:
        void walk() { /*犬の歩き方*/ }
}
```

Animalクラスを継承することで、必ずwalk()という関数をオーバーライドして実装するという制限を付けられる。

インターフェースを共通にするために用いられる！

ポリモーフィズム

インターフェースが共通化されたライブラリを実現するために用いられることが多い手法

- 純粋仮想関数を用いて実装が必要な処理をライブラリ利用者に提示する
- アップキャストを用いて同じ基底クラスをもつオブジェクトに対する処理を共通化する

OpenCV、Qtなどのライブラリで用いられている。

ポリモーフィズムを用いた大規模なプログラムを構成するための方法論として「デザインパターン」が広く知られている。

テンプレート

関数テンプレート

引数や返り値の型を指定せずに関数を実装する

```
template <class T>
T MAX(T x, T y) {
    return x < y ? y : x;    //x, yの大きい方を返す
}
```

使い方

```
int i = MAX(1, 2);           //整数を引数にする
char c = MAX('a', 'c');      //文字を引数にする

int i = MAX<int>(1, 2);      //型を明示化して記述できる

int i = MAX(1, 'c');         //複数の方が混ざっているとエラーになる
int i = MAX<int>(1, 'c');    //型を明示化すれば'c'は整数とみなされエラーは出ない
```

クラステンプレート

メンバ変数や、メンバ関数の引数、返り値の型を指定せずにクラスを実装する

```
//配列クラス
template <class T>
class Vector {
    private:
        T data[100];            //データ保持領域
        int size;               //追加されているデータのサイズ
    public:
        Vector() : size(0) {}   //コンストラクタ
        void append(T arg) {    //末尾に追加する関数
            data[size] = arg;
            ++size;
        }
}
```

使い方

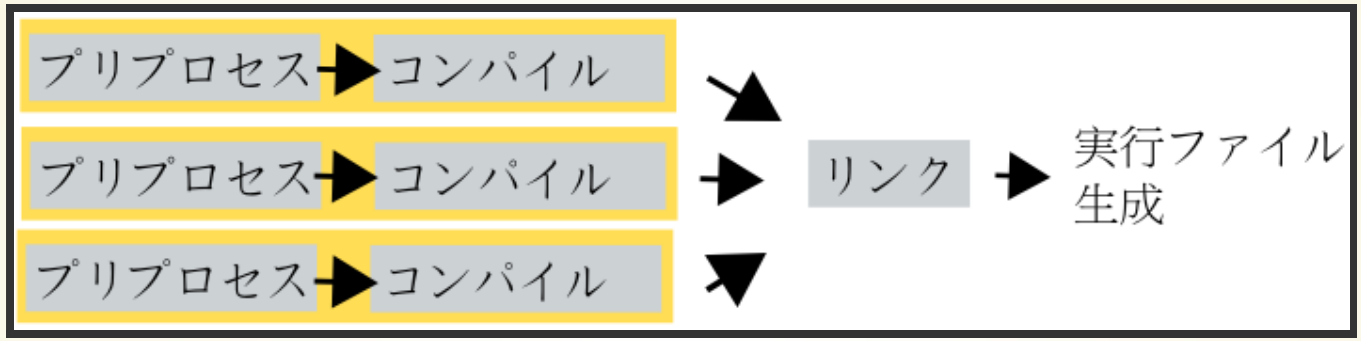
```
Vector<int> vint;
vint.append(4);

Vector<std::string> vstr;
vstr.append("This is a pen");
```

型ごとにクラスを定義しなくても様々な型に対応することができる

プログラムが出来るまで

「ビルド」の流れ

複数のファイルに記述されたプログラムから，実行ファイルを生成するまでの操作を「ビルド」という



- プリプロセス
  - コンパイル前のコードの貼りあわせ，置き換え．「#include」プリプロセッサディレクティブなどが行っていること．
- コンパイル
  - コードの意味を解釈して一時ファイルを作成する
- リンク
  - 複数の一時ファイルをつなぎあわせて実行ファイルを生成する．静的リンク，動的リンクが有る．

翻訳単位

- コンパイラがコンパイルするときに把握している範囲
- あるファイルとそれがインクルードしているファイルが翻訳単位となる.

```
//inc.h
int a;
```

```
//main.c
#include "inc.h"

int main(void) {
    std::cout << a << std::endl;
}
```

上の例では翻訳単位は「main.c」と「inc.h」

- コンパイルはクラスや関数の宣言があればできる
- リンクはクラスや関数の実装を見て行われる.
- 何をインクルードファイルに記述するか？

実践的ファイル構成

- インクルードファイルに記述すること

```
//test.h
#ifndef TEST_H  //2重インクルード防止のためのコード
#define TEST_H

class Test {     //クラスの宣言
    int a, b
    int get();
}

int somefunc(int);     //メンバ関数でない関数の宣言(関数プロトタイプ)

#endif
```

- .cファイルに記述すること

```
//test.c
#include "test.h"

Test::Test() {}

int Test::get() { return a; }

int somefunc(int a) {
    return 132132;
}
```

一般的にクラスの名前と同じ名前のファイル名で管理する

END