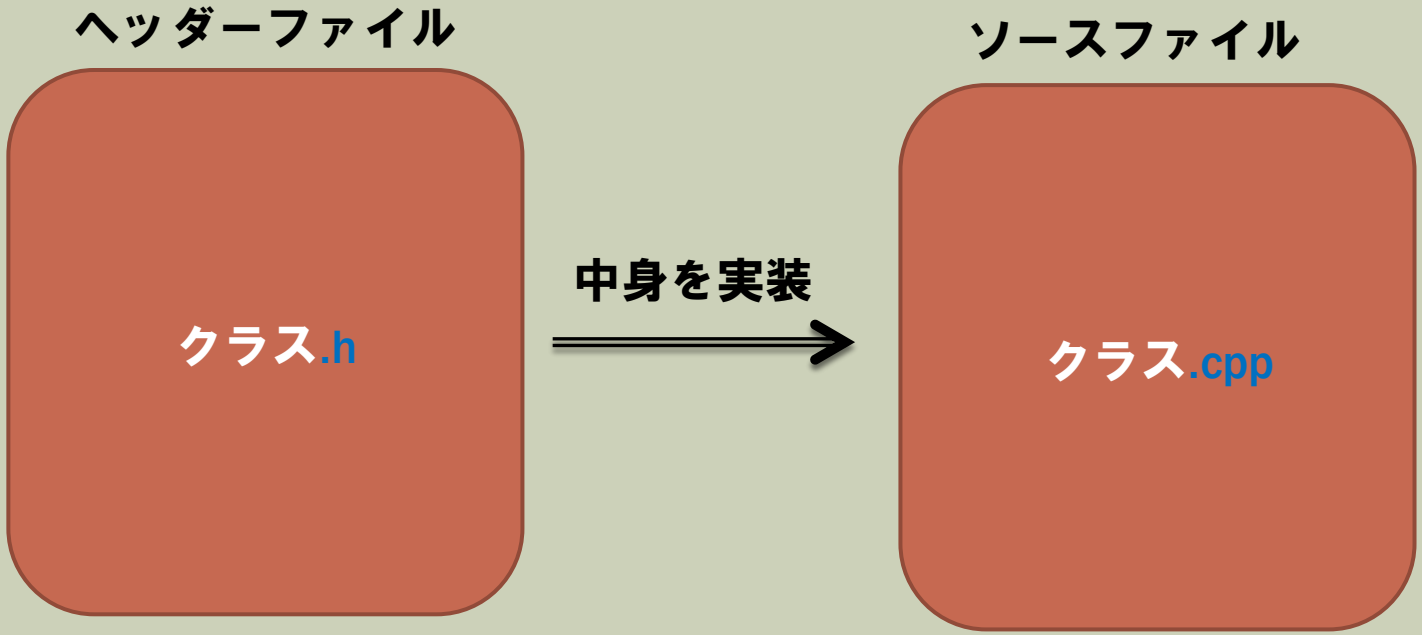
C++　CLASSとか。

@yasyoku_

クラスを作ろう

- とにもかくにもクラスを作ります。
- ポピュラーな書き方は以下です。



こんなかんじ。

- hito.h

```
class hito{
public:
 int age;
 void print_age();
}
```

- hito.cpp

```
#include”hito.h”

void hito::print_age(){
 std::cout << age;
}
```

コンストラクタ

- クラス名　変数名 = クラス名();

  ↑こいつ

- メモリ確保したり初期化したりする。

こんなかんじ。

- hito.h

```
class hito{
public:
 int age;
 hito();
}
```

- hito.cpp

```
#include”hito.h”

hito::hito(){
        age = 0;
}
```

コンストラクタとオーバーロード

- みんな0歳な訳ないです。自分で年齢設定したいです。

  - hito() {
    age = 0;
    }

  - hito(int settei_age){
    age = settei_age;
    }

- こういう風に定義できます。同じ名前だけど引数によって呼び出すモノを変えてくれる。**オーバーロード**っていいます。コンストラクターにも使えるしメソッドにも使えます。

デストラクタ

- **デストラクター**というものがあります。const-(作成する等)に対しde-(破壊する)みたいな感じです。
- こいつは**インスタンスが破棄**されるときに勝手に呼び出されます。自分で呼ぶこともできます。

```
hito() {
 age = 0;
 std::cout << “コンストラクタです。¥n”;
}
```

```
~hito() {   std::cout << “デストラクタです。¥n”;
}
```

- こんな感じで定義するとわかりやすい。

デストラクタ

```
void test() {
    hito Human = hito();
}
```

- これをmain文の中で実行してみると判ります。
- メソッドが終了するとインスタンスは破棄されるためデストラクタが呼び出されます。

コピーコンストラクタ

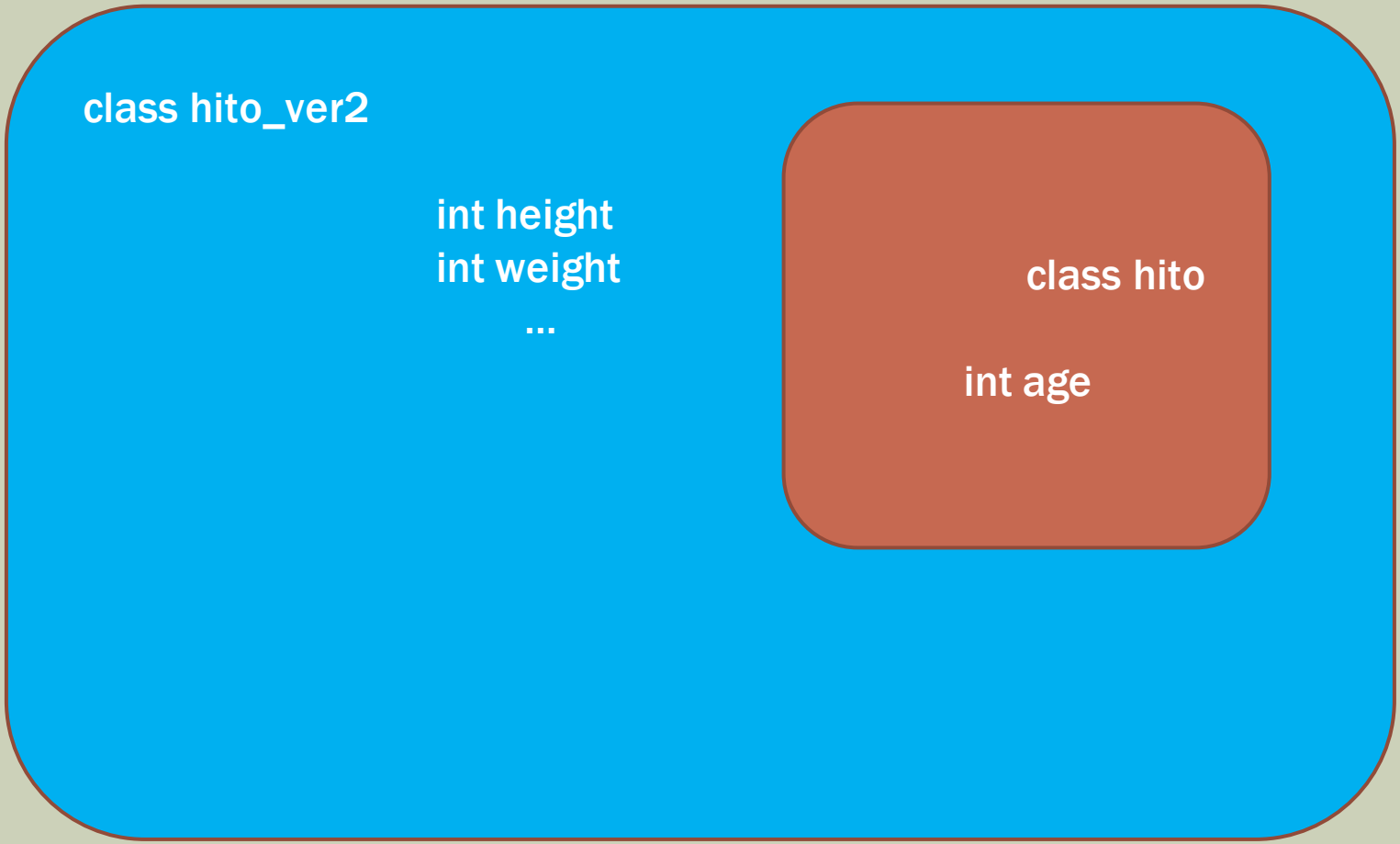
```
hito(const  hito& obj){
    std::cout<< “コピーコンストラクタだよ¥n”;
}
```

- これだけの定義で**コピーコンストラクタ**は定義できます。

```
hito You = hito();
hito Newman = You;
```

- こんなときに呼び出されます。
  代入と違うのは**メモリーを新しく確保して、情報をコピー**することです。代入だと両方のインスタンスが同じデータを参照してしまう可能性があります。

継承



継承

- class hito{
  ...
  }

- class hito_ver2 : public hito {
  ...
  }

- こんな感じで書けば継承される。更に新しい機能を追加したりする場合とか、継承する。

継承したときのコンストラクタ

```
hito_ver2(int  age, int height){
    this->age  = age;
    this->height  = height;
}
```

- って書ける。変わらないね。

装飾子

```
class hito{
public:
 ~~~~
 ~~~~
}
```

- このpublicとはなんぞや？というお話。

装飾子

装飾子	派生クラスからのアクセス
private	できない
protected	できる
public	できる

装飾子	全く関係無い所からのアクセス
private	できない
protected	できない
public	できる

privateな変数はコンストラクターやpublicなメソッドで操作する

STATIC(静的)

- staticな変数を作ります。staticな変数はインスタンスを作成してもすべてのクラス間で共通となります。
- 定義の仕方としては　static 型名 変数名;　となります。
- インスタンスを作成しないでも参照でき、クラス外から参照するときは

クラス名.変数名

- となります。

オーバーライド

```
class hito{
~~~
void print();
}
```

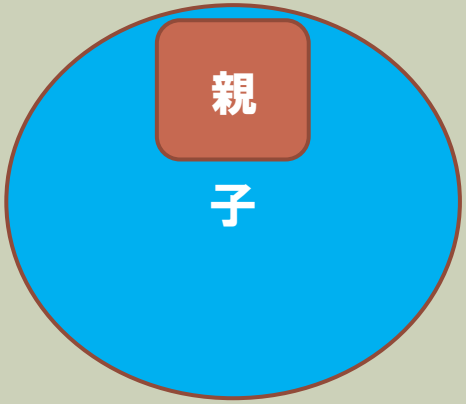
- こんな関数があったとする。子クラスでもprintって関数を作りたい。でも機能変えたい！！ってときに使うのが**オーバーライド**

```
class hito_ver2 : public hito{
~~~
void print();          ←これだけでオーバーライドになります。
}
```

- 正直別名の関数にすればよくね？ってなりますけど、次の機能に活かされます。

ポリモーフィズム

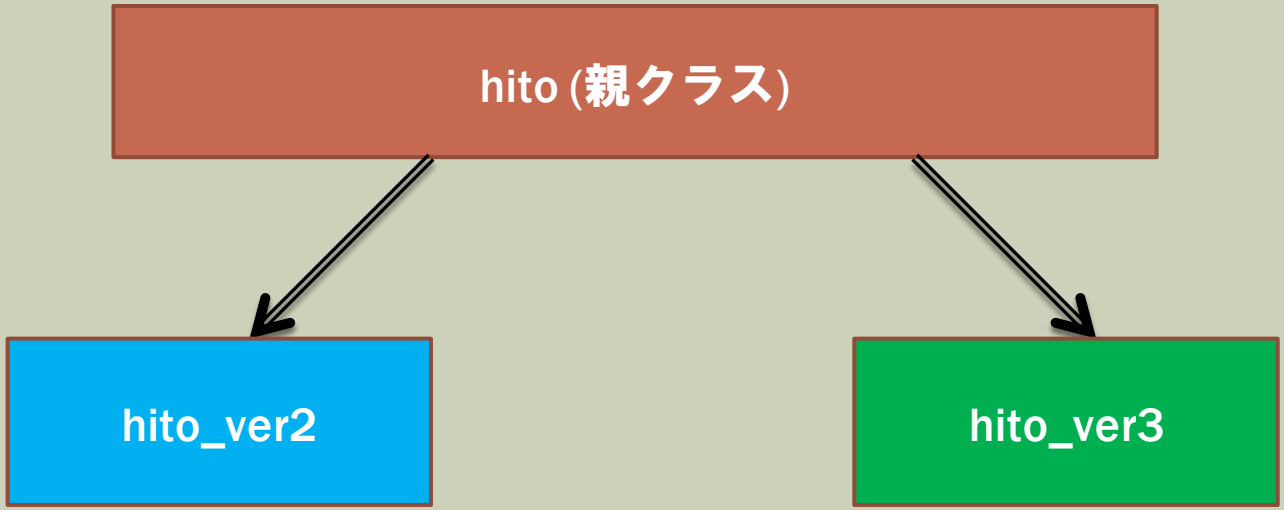
- **親クラス ⊆　子クラス**



- なので、親クラスの型に子クラスいれたら機能的には収まりますよね？ね？
  - `hito human = hito_ver2();`
- こんなことが出来るんです。hitoの範囲内であればhito_ver2の機能は実装されているはずなので動かせる、のです。

下準備

```
class hito_ver3 : public hito{
public:
void print();          //中身はなんでもいいです。
}
```



配列化

- hito humans[3] = { hito(), hito_ver2 (), hito_ver3 () };

- こんなこともできちゃいます。このとき、humans[0~2].print()とやるとhitoのprintだったりhito_ver2のprintだったりhito_ver3のprintだったり……。って期待しますが、今の実装では動かないです。

- 実際はアップキャストという方法を使います。

アップキャスト

- hito *TEST[2];
- TEST[0] = new hito(); //動的にメモリ確保
- TEST[1] = new hito_ver2(); //動的にメモリ確保

- こんな風に書くことができます。
  子を親に突っ込みました(アップキャスト)

- アップキャストを行うときは必ずポインターで行って下さい。
  行わない場合はただの親クラスとして扱われてしまいます。

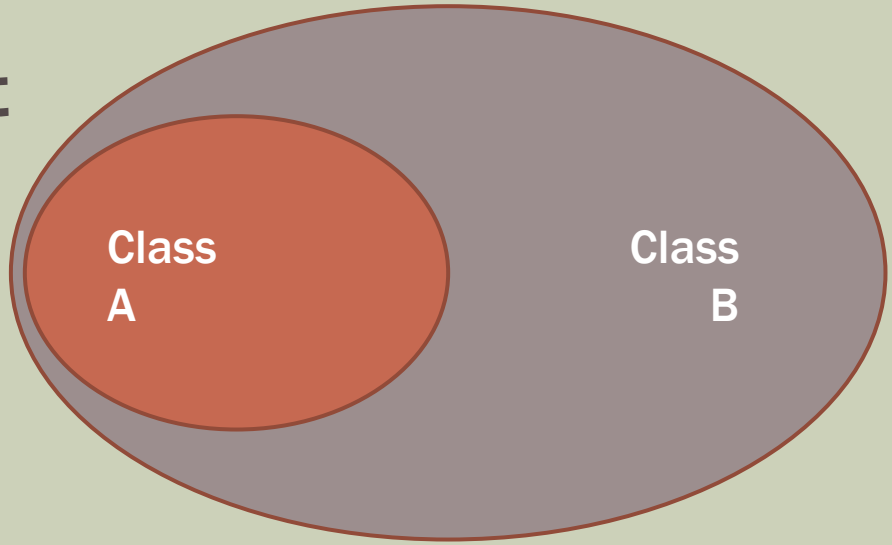
- newしたらdeleteするのも忘れずに。

アップキャストの注意点

- 親クラスをA、子クラスをBとする。

AにA	可能
AにB	可能
BにA	不可能
BにB	可能

- BにAを入れると$\bar{A} \cap B$の部分に対応できない！！



アップキャストとオーバーライド

- アップキャストとオーバーライドを行うことによる利点。
  以下の場合の実行結果が変化します。

```
Creature *TEST[2];
TEST[0] = new Creature();
TEST[1] = new Human();

TEST[0]->print();
TEST[1]->print();
```

```
このCreatureクラスのnameはnullです
このHumanクラスのnameはnullだよ
続行するには何かキーを押してください . . .
```

- こんな感じでオーバーライドした関数が自動的に呼び出される

- ちなみにvirtualを抜いたときの実行結果が次のやつ。

```
このCreatureクラスのnameはnullです
このCreatureクラスのnameはnullです
続行するには何かキーを押してください . . .
```

動かすゾ

- これでhumans[0]はhitoのprint、humans[1]はhito_ver2のprintが出るんでしょ！？！？

```
humans[0]->print();
humans[1]->print();
```

```
0歳
0歳
```

- って思いますが、こうなって出ません。原因としてはオーバーライドした場合でも**親クラスの関数を優先して使用します**。

優先順位の変更(VIRTUAL)

- なら子クラスに同名の関数が実装されたとき、子クラスの関数を優先して実行して欲しい。っていうコマンドを付けます。

```
class hito {
~~~
virtual void print();
}
```

- こうするとアップキャストした場合子クラスではオーバーライドされた関数を優先して使います。