

C++を始めよう

C++は、オブジェクト指向の言語だけど、
オブジェクト指向がそもそもわからない人も多いと思うので、
まずは、中身の前に概念から説明しよう

C言語を勉強してきたみんなにとってのメリットはC言語と互換性があり、
ソースコードを再利用できる点が挙げられる。
変数の宣言や配列など、様々なことがC言語と同じか、それをベースにしている。
その上で、C++は、オブジェクト指向という考えを取り入れているんだ。

で、オブジェクト指向って何？という最初の話になるわけだと、
まず、オブジェクト指向の、オブジェクト(Object)とは、英語で「モノ」を表す言葉です。
ちょっとわかりにくいかな？
装置と考えると分かりやすいかもしれない。
例えば、電子レンジの仕組みを知らなくても電子レンジは使えるよね？
でも、電子レンジの構造を理解していれば、固形物を温める際は、
電子レンジは固体より液体の方が分子振動が起きやすいから、
固形物を温める際は、水を少しつけてあげている人もいます。
中身を知っていれば、知っているなりに知らなければ知らないなりに使うことが出来る。

プログラムに戻して考えてみよう。
関数の中身は理解していなくても呼び出して使うことが出来るというイメージだ。
今までの授業でも、先生が用意した関数を利用して、色々作っていたと思う。
そうした用意されたものや自分で用意したものを、
利用してプログラムしていくのが、C++だ。
でも、それだけだと今までと同じで、何が違うんだろう？って思うよね。

今までは、変数やそれを利用する関数はそれぞれバラバラで用意していたと思う。
C++では、それらを1セットとして、扱うんだ。
これをクラスという。
クラスは基本クラス(基底クラス)と派生クラスとある。
基本クラスは、超簡単に言ってみれば、
色々なベースとなりそうなものだけが、集まったクラスだ。
例えば、画像の読み込み、その画像の表示・アニメーション、移動など、
自機や敵、弾など関係なく共通の部分だけを集めたと思えばいい。
じゃあ、派生クラスはというと、そこから、自機なら自機の、
敵なら敵の専用の部分を付け足したものと考えれば、理解しやすいと思う。
この付け足すにあたって、C言語なら、付け足す以外の部分も全部書か、
書かない場合でも、汎用関数で利用する変数(構造体)を、
それぞれ宣言しないといけないなど、巨大なものを作ろうとすると、
かなり複雑になってしまう。
C++場合では、共通部分は基本クラスを見れば済んでしまう。
派生クラスの中でいじる分には、ほかに影響しないので、
複数人でのプログラムもしやすいはずだ。

と、いった感じがC++になるんだけど、雰囲気だけでも感じれたなら、ここでは十分だ。

C++の世界によろこそ！

ということで、定番のHelloWorldを表示させよう。

```
1 #include "stdafx.h"
2 #include <iostream>
3 int main() {
4     int a;
5     std::cout << "HelloWorld." << std::endl;
6     std::cin >> a;
7     return 0;
8 }
```

見慣れた部分と見慣れない部分とある感じかな？

まず、includeの中<>内は、C++のコーディングでは、.hを書かない。

入出力関連ヘッダの場合は、iostreamを記述すればいい。

この辺は、C言語同様に、使用するもので何をインクルードするのは、調べて記述しよう。

そして、5行目だが、見慣れないものだらけだね。

std::は後で説明するとして、まずはその次の文字coutを見てみよう。

これは何かというと、printfのC++版なんだ。

ちなみにcoutではなく、シーアウトだからね？

で、文末に\nをつけるのと同じ感じで、つけるのが、endlだ。

ちなみに読み方は決まってないので、好きに呼んでくれ、エンドエルとか。

で、<<の記号だが、C++言語の入出力ストリームでは、<<(>>)を用いることにより、ストリームと呼ばれる対象に対するデータのやりとりを行うんだ。

代入というよりは、渡すというイメージが近い。

後ろから読んでいくと、“HelloWorld.”に文字のエンドを付け加えて(endl)、

文字出力(cout)にその文字列を渡す、結果、画面に表示されるといった感じだ。

ちなみに、5行目をC言語風に書くと、次のようになる。

```
printf("HelloWorld.\n");
```

わかったところで5行目を読んでみよう。

なんて読むかな？

シーインって読むんだ。

何をするかというとキー入力をさせたいときに使うんだ。

C言語でいうscanfみたいなもんだね。

ということで、このプログラムは、簡単に言えば、ハローワールドを表示して、

キー入力を待って、終了するということになる。

さてさて、ひと通り説明したところで、std::の説明をしようか。

std::は名前空間と呼ばれるものを表している。

名前空間って聞きなれない単語だね。

名前空間というのは、特定のルールの通用する範囲の名前といえばどうだろうか。

家庭とか学校とかそこでしか通じないローカルルールってあるよね？

そのローカルルールが通じるエリアの名前と思ってくれたらいい。

これのいいところは何かというと、同じ名前のルールでも、名前空間が違えば、

違うものとして存在していいってことなんだ。

なので、C言語と同じ名前の変数名は存在できなかったと思うけど、

C++言語の場合、名前空間が違えば、同じ変数名でも存在できるってことになる。

ちなみに単一の名前空間を使う場合は、いちいち全部に、

std::とつけるのが面倒なので、次のような内容を、includeの後に書くことが多い。

```
using namespace std;
```

そうすれば、5行目は次のようになる。

```
cout << "HelloWorld." << endl;
```

これで、君もC++言語の入り口に立ったぞ！

クラスを理解しよう

前は『HelloWorld.』を表示したわけだけど、今回はクラスについて話していくよ。
オブジェクト指向の言語では、必ず登場するクラス。
要はC++のモノという話を前回したわけだけど、そのモノがこのクラスと思ってくれればOK。
さて、C++のクラスとはどんなものだろうか？
今までみんなが勉強してきた中では、構造体が結構近いかな。
構造体は、特定の目的のために集めた変数の集合体だったよね？
クラスはそれに加えて、関数まで足したものと言えれば伝わるかと思う。
そのクラスに含まれるものをメンバと呼び、
それぞれを、メンバ変数、メンバ関数と呼びます。

《オマケの話》

ちなみにメンバーじゃないの？と思うかもしれないけど、この辺は、過去のマイクロソフトのせい。
マイクロソフトが呼び方で、伸ばす棒を付けないのを標準としたんだ。
だから、プログラマとかコンパイラとか呼ぶ呼び方が定着している。
けど、この話には落ちがあって、今のマイクロソフトは逆に、伸ばす棒を付ける方を標準としている...。
なので、どちらでも良いので、入った会社とか相手に合わせて、

では、実際にクラスの宣言の部分を示してみよう。

<<sample.h>>

```
#ifndef _SAMPLE_H_
#define _SAMPLE_H_


// クラス宣言
class CSample
{
public:
    void set(int num);
    int get();
private:
    int m_num;
};


#endif // _SAMPLE_H_
```

クラスの宣言の際には、まずclassと書いて、その後ろにクラス名を記述する。
構造体同様に{}でくぐり、その中に、メンバを記述していく。
この際、public: やprivate: というのが出てくるけど、これは、アクセス修飾子と言うんだ。
何かというと、メンバの公開範囲を示したものだ。
他のクラスからも見えてよいものはpublicに記述し、見えたら困るものはprivateに記述する。
クラスは、関数を内包するから、その関数で使用している変数で、他から値を弄られると困る変数とかあるよね。
そういった変数をprivateにしたりするんだ。
その変数にアクセスする場合は、アクセス用の変数を用意して対応する形になる。
例えば、キャラの座標用のメンバ変数をprivateで用意し、GetPos()みたいな感じで座標を取得する。
書き換えたい場合は、SetPos()みたいな感じ。
ただ、この辺はやりすぎると不便さが出てくるので、考えて設計をしてほしい。

じゃあ、続いて中身を書いていこう。

```
<<sample.cpp>>
#include "stdafx.h"
#include "sample.h"

void CSample::set(int num)
{
    
}

int CSample::get()
{
    
}
```

さあ、空欄の個所を埋めてみてほしい。
setの方は、int m_numに値を書き込む。
getの方は、同じ値を取得する関数だ。

答えは、こうなる。

```
void CSample::set(int num)
{
    m_num = num;
}


int CSample::get()
{
    return m_num;
}
```


さて、クラス側の準備が出来たところで、それを呼び出すmain側を作ってみよう。

```
#include "stdafx.h"
#include <iostream>
#include "sample.h"

using namespace std;

int main()
{
    CSample obj; // CSampleをインスタンス化
    int num;

    obj.set( num ); // CSampleのメンバ変数をセット
     // メンバ変数の値を出力

    return 0;
}
```

クラスを使う際は、構造体や変数を使うのと同じようにすればよい。
その上で、setのメンバー関数を呼び出そう。
呼び出す際は、インスタンス名の後ろに **.メンバ関数名** で呼び出せる。

他の行は、

- ・数字の入力を促すメッセージを表示する
 - ・数字の入力をさせる
 - ・入力された数字を表示する。
- が入るので、それも入れてみよう。

では、答え合わせ。

```
int main()
{
    CSample obj; // CSampleをインスタンス化
    int num;

    cout << "整数を入力して下さい:" << endl;
    cin >> num;

    obj.set( num ); // CSampleのメンバ変数をセット
    cout << obj.get() << endl; // メンバ変数の値を出力

    return 0;
}
```

では、下記のように、このプログラムでインスタンスを2つ行うとどうなるでしょうか？
今までの感覚だと、obj1の表示もobj2の表示も同じになると思わない？

```
int main()
{
    CSample obj1; // CSampleをインスタンス化
    CSample obj2; // CSampleをインスタンス化
    int num;

    cout << "整数を入力して下さい:" << endl;
    cin >> num;

    obj1.set( num ); // CSampleのメンバ変数をセット
    obj2.set( num ); // CSampleのメンバ変数をセット
    cout << obj1.get() << endl; // メンバ変数の値を出力
    cout << obj2.get() << endl; // メンバ変数の値を出力
    return 0;
}
```

それぞれインスタンスされれば、丸ごとメモリー上に生成されるので、
変数の内容は影響を受けないんだ。

さて、クラスについての話をしたわけだけど、前回のは、ユーザーが作ったクラスなので、そのまんま、ユーザークラスと呼ぶ。
その一方で、C++に標準で入っているクラスもある。
例えば、stringクラスなどがある。
実際に使ってみよう。

```
#include "stdafx.h"
#include <iostream>
#include <string>

using namespace std;

int main() {
    int tmp;
    string s;
    s = "今日の天気は、";           // 最初の文字列
    s.append("晴れ。");             // 文字列の追加
    cout << s << endl;
    cout << "文字列の長さ:" << s.length() << endl;
    // printfで表示
    printf("%s\n", s.c_str());
    cin >> tmp;
    return 0;
}
```

C言語で、文字列の長さを調べて返す関数を作ることは出来ると思うけど、C++では、このようなクラスも用意されている。
この様な標準クラスを使うことで、開発効率が上がるので、使っていこう。
なお、標準クラスだけでは不十分なので、環境にもよるが、テンプレートを使えると、なおいい感じだ。
vectorなどが良く使われるので、ここでは解説しないが出来れば、マスターしておこう。

コンストラクタとデストラクタ

さて、クラスのメンバー関数で、色々出来ることはわかったと思う。

C++では、特別な役目のあるメソッドも存在している。

まずその前に、いくつかを次のように書き換えよう。

```
int main()
{
    CSample *obj1 = NULL;
    int num;
    obj1 = new CSample;           // CSampleをインスタンス化

    cout << "整数を入力して下さい:" << endl;
    cin >> num;

    obj1->set( num ); // CSampleのメンバ変数をセット
    cout << obj1->get() << endl; // メンバ変数の値を出力
    cin >> num;
    return 0;
}

void CSample::set(int num)
{
    m_num += num;
}
```

m_numに値をどんどん加算していくようにする感じだ。

で、ここで、アレ？大丈夫なの？ふと気づいた人は偉い！

何かというと、m_numの初期値が不明だから、どんな値になるかわからないんだ。

実行した結果、変な数字が入っている人も多い。

ということで、初期化が必要になるわけだけど、今までのようにinit関数を作るのかというと、そうではない。

そういうのが必要なシチュエーションもあると思うけど、

メモリー上に確保された時点で、実行される特殊なメソッドがある。

これは、メイン関数に呼び出しを書く必要が無く、自動で実行される。

今回は、それを使ってみる。

```
CSample::CSample():m_num(0)
{
    cout << "メモリーに追加" << endl;
}
CSample::~CSample()
{
    cout << "メモリーから削除" << endl;
}
```

上記2つのメンバーを追加しよう。

上の方は、コンストラクタといい、生成時に実行される。

下の～が付いている方をデストラクタといい、削除時に実行される。

今回は、説明用にメッセージを表示するようにしている。

コンストラクタで変数を単純初期化するばあいは、

:の後ろに変数名と()内に初期化数値を書けばOKだ。

複数ある場合は、「,」でつなげて書く感じになる。

なお、初期化は普通に{}内に書いても問題ない。

特に初期化時に条件判断などはいる場合は、そうしないと書けないしね。

さて、動的に生成する際に、newを使ったけど、C言語ではmallocを使ったかと思う。
C++ではmallocは使わず、newを使っていく(削除時はdelete)

では、先ほどの内容のメンバー変数のint m_numのところを、int * m_numにして、
必要な処理を追加してみよう。
それが、出来たら、m_numを配列の動的確保に変えて、
入力数値をそのまま保存して、入力数値の合算値を表示できるようにしてみよう。

変数をnewする際はこう書く。

```
m_num = new int();
```

削除時は、こうなる。

```
delete m_num;
```

配列の場合は、たとえば10個なら、

```
m_num = new int[10];
```

この様に書き、削除時は、

```
delete []m_num;
```

のようになる。

静的メンバ

前回までにやった内容のメンバーは、インスタンスメンバーといい、要するに、インスタンスして初めて使うことができるから、そう呼びます。一方で、インスタンスしなくても使えるメンバーもあります。それは、静的メンバといい、次のように記述します。

```
static ○○○;
```

C言語でも同様に記述しましたが、同じです。

では、実際に使ってみよう。

```
int get();
```

```
int m_num;
```

この2つを静的メンバにしてみよう。

呼び出し方は、

CSample::get(); のように書けばよい。

obj1->get(); のように書いても良いが、

静的メンバの場合、上記のように書くことが多い。

ここで、どのインスタンスのメンバーかわからないじゃないと思った人もいるかと思う。

静的メンバは、プログラムが実行された時点で、メモリ上に確保される。

インスタンスメンバーと違い個別に持つことはない。

なので、インスタンスを区別する必要はないので、先のような書き方となるわけだ。

じゃあ、静的メンバの制約が無いかというと、ある。

簡単に言えば、インスタンスメンバにアクセスすることができないんだ。

なぜか、変数も関数もすべてアドレスで管理されていることを思い出してほしい。

つまり、インスタンスして初めてアドレスが割り振られるインスタンスメンバの、

変数や関数のアドレスにはアクセスのしようがないということだ。

逆に、インスタンスメンバからは、静的メンバは、アクセスできる。

理由はわかるよね？

そう、アドレスが分かっているからだ。

そこが理解できれば、静的メンバを使いこなせるはずだ。

…個人的にはあんまり使うことはないと思うが。

継承

さて、いよいよC++の重大要素の一つ、継承をやっていく。
ちなみに、英語で言うとinheritanceという。
で、ここでいう継承とはどういうことだろうか？
よく、例えで使われるのは、車だ。
まず、基本的な車をイメージしてほしい。
機能として、人が乗れて、タイヤがあってハンドルがあって…と。
実際に世の中の車をみると、通常の乗用車以外にも、クレーン車や、救急車など、
様々な機能を持った車が存在している。
その基本的な車が継承元であり、機能拡張された車が継承されたモノと考えればOKだ。

さて、話をC++に戻そう。
この継承元となるクラスのことを親クラスという。
もしくは、スーパークラスという。
余談だが、拡張されたものが凄いのは当たり前、元になったやつが凄いんだという
海外では当たりの考えがこういうところにも出てるんだなと思う。
そして、機能拡張されたモノを子クラスという。
もしくは、サブクラスという。

おさらいで、まず親クラスを書いておこう。

```
<<sample.h>>
#ifndef _SAMPLE_H_
#define _SAMPLE_H_

// クラス宣言
class CSample
{
public:
    CSample();
    ~CSample();
    void set(int num);
    int get();
private:
    int m_num;
};

#endif // _SAMPLE_H_

<<sample.cpp>>
CSample::CSample():m_num(0)
{
    cout << "親をメモリーに追加" << endl;
}
CSample::~CSample()
{
    cout << "親をメモリーから削除" << endl;
}
void CSample::set(int num)
{
    m_num = num;
}
int CSample::get()
{
    return m_num;
}
```

さて、次は、それを継承したクラスを書いていく。

<<child.h>>

```
#ifndef _CHILD_H_
#define _CHILD_H_

// クラス宣言
class ChildCls :public Csample;
{
public:
    ChildCls();
    ~ChildCls();
    void add(int num);
    int sum(void);
private:
    int s_num;
};

#endif // _CHILD_H_
```

<<child.cpp>>

```
ChildCls::ChildCls():s_num(0)
{
    cout << "子をメモリーに追加" << endl;
}
ChildCls::~ChildCls()
{
    cout << "子をメモリーから削除" << endl;
}
void ChildCls::add(int num)
{
    s_num += num;
}
int ChildCls::sum(void)
{
    return s_num;
}
```

<<main.cpp>>

```
#include "sample.h"
#include "child.h"

int main() {
    ChildCls *obj1;
    int num;
    obj1 = new ChildCls;

    for(int j = 0; j < 5; j++){
        cout << "整数を入力して下さい:" << endl;
        cin >> num;
        obj1->set( num );
        cout << "入力値:" << obj1->get() << endl; // メンバ変数の値を出力
        obj1->add( num );
    }
    cout << "合計値:" << obj1->sum() << endl; // メンバ変数の値を出力
    cin >> num;
    delete obj1;
}
```

さて、中を見ていこう。

クラス宣言をする際、親の場合は、クラス名を書けばよかった。

継承する場合は、これの後ろに継承元のクラス名を書けばOKだ。

main関数を見ればわかるが、継承したクラスをインスタンスし、実行した場合、親クラスで記述した内容も実行されるし、使うことが出来るんだ。

なので、例えば、移動や当たり判定などの基本処理を、親クラスに入れ、プレイヤーや敵によって違う部分は子に入れるイメージだ。

継承の際の実行順で気を付けるべきはコンストラクタとデストラクタの部分だ。

コンストラクタは、親→子の順で実行され、デストラクタは子→親の順で実行される。

継承の継承も可能だが、コンストラクタとデストラクタがややこしくなるので、気を付けよう。

最後に、C++言語では、継承を用いる場合、

デストラクタにvirtual(バーチャル)修飾子に関しては、必ずつけるようにしてください。

この修飾子については、ここでは、詳細は触れません。

protected修飾子

さて、クラス宣言の際にpublic、privateがあることは説明したが、もうひとつある。それは、protectedと呼ばれるものだ。プロテクト、保護されたものということだが、要するに、クラスに保護され、他からは触れないものだ。ん？ それだと、privateとどう違うんだ？と思ったかもしれない。privateは外から触れない、中からだけだ。一方で、protectedは確かに外からは触れないという点では、privateと同じだが、継承した子クラスからは、public同様にアクセスすることが出来る。表にするとこんな感じだ。

		親	子	外部
親	public	-	○	○
	private	-	×	×
	protected	-	○	×
子	public	-	-	○
	private	-	-	×
	protected	-	-	×

演習として、親クラスのコントラクタの以下の部分をinit()のprotectedメンバー関数として作成し親及び子クラスのコントラクタから呼び出すプログラムに変更してみよう。

```
cout << "メモリーに追加" << endl;
```

変更出来たら、mainからも呼び出してみよう。こちらは、コンパイルエラーが出ることが確認できればOKだ。

ポリモーフィズム

日本語で簡単に言おうとすると、『多様性』、『多態性』、『多相性』、となるんだが、これだけでわかるなら、苦労は無い。

例えるなら、こういうことだ。

『再生ボタンのスイッチを押す』と聞いて何をイメージしたかな？

音楽の再生ボタンを押す、DVDなどの再生ボタンを押す、youtubeの再生ボタンを押すなど、色々な『再生ボタンのスイッチを押す』があるよね。

この単一の『再生ボタンのスイッチを押す』というアクションに対しての多様性を、ポリモーフィズムという。

再生ボタン自体に音楽の再生機能が備わっているわけではなく、

音楽プレイヤーというオブジェクトにある再生ボタンの場合、音楽の再生機能があり、

DVDプレイヤーというオブジェクトにある再生ボタンの場合、DVDの再生機能がある。

つまり、再生ボタンにより起こる現象はそれを備えているオブジェクトに依存するということが言分かったかな？

オーバーロードとオーバーライド

では、このポリモーフィズムの考え方を理解したところで、その概念から生まれたオブジェクト指向の言語特有の記述をやっていこう。

まずは、オーバーロードから。

前回、addのメンバー関数を追加したけど、1だけ足したい場合に、いちいち引数を入れるのがその場合、C言語なら、新たにincrement()といった感じで別に関数を作ったと思う。

でも、これをポリモーフィズムの考え方で実装する場合、次のようになる。

```
void ChildCls::add(int num)
{
    s_num += num;
}
void ChildCls::add()
{
    s_num ++;
}
```

クラスメンバーに下記も足しておこう。

```
int add();
```

さて、この内容、C言語なら、同じ名前で引数が違うだけだと、まずコンパイルが通らない書き方でも、C++の場合は、ポリモーフィズムの考え方で、これが許容される。

区別はどこでしているかというと、引数や戻り値で行っている。

なので、逆に言えば、引数や戻り値が、全く同じで、処理だけ異なるものを複数書くことは出来ないともいえる。

さて、他にもやってみよう。

コンストラクタで、現在、メッセージを表示するようにしているが、これの有り無しを呼び出し側で制御したい。

引数無しの場合は表示されず、引数ありの場合は、表示ON・OFFを切り替えれるようにして欲

さて、続いて次は、オーバーライドをやっていこう。
で、オーバーライドとは何か？
ライドは乗るとか積み込むとかそういう感じだ。
ということは、オーバーライドとは上に積んでいくという感じになる。
プログラムの際には親のメンバー関数の上にかぶせて全く同じものを子供で宣言するということ
では、実際にやってみよう。
親にview()というメンバー関数を追加しよう。
中身は、入力値:○という感じで、m_numの値を画面に表示してほしい。
出来たら、メインを次のようにしてみよう。

```
int main() {
    ChildCls *obj1;
    Csample *obj2;
    int num;
    obj1 = new ChildCls;
    obj2 = new Csample;

    for(int j = 0; j < 5; j++){
        cout << "整数を入力して下さい:" << endl;
        cin >> num;
        obj1->set( num );
        obj2->set( num );
        obj1->view(); // メンバ変数の値を出力
        obj2->view(); // メンバ変数の値を出力
        obj1->add( num );
    }
    cout << "合計値:" << obj1->sum() << endl; // メンバ変数の値を出力
    cin >> num;
    delete obj1;
}
```

さて、この時点では、親と子のどちらから呼び出したview()も表示されるものは同じだ。
じゃあ、次は、子に親と同じようにview()をメンバーに追加し、
中身は、現在の合計値:○という感じでs_numを表示するようにしてみよう。

そうすると実行結果は怎么样了らうか？
親から呼び出した場合は、入力値が表示されて、
子から呼び出した場合は、現在の合計値が表示されたかと思う。
これが、オーバーライドの効果だ。

使い方としては、例えばキャラクターのインプットのメンバー関数があったとして、
2人プレイで、人がプレイする場合はキーからの情報を取得して処理するようにし、
人がプレイしない場合は、AIから取得するように初期化時に
その部分だけ違う子クラスを用意する感じだ。

C++の細かいところ

さて、継承やポリモーフィズムの考え方など大きなところをやったところで、実際に書いていくと、ぶち当たるところを少し補足しておく。

まず、よくあるのがクラスの相互参照によるコンパイルエラーだ。実際に例を示すのでやってみよう。

ClassA.h

```
#ifndef _CLASS_A_H_
#define _CLASS_A_H_

#include "ClassB.h"

class ClassA{
    ClassB* m_ClassB;
    ClassA();
    ~ClassA();
};

#endif // _CLASS_A_H_
```

ClassB.h

```
#ifndef _CLASS_B_H_
#define _CLASS_B_H_

#include "ClassA.h"
class ClassB{
    ClassA* m_ClassA;
    ClassB();
    ~ClassB();
};

#endif // _CLASS_B_H_
```

上記の例では、ヘッダーを見てみると、ClassA.hでClassB.hをインクルードしています。さて、次にインクルードされたClassB.hを見てみるとClassAがインクルードされています。さてさて、次にインクルードされたClassA.hを見てみるとClassBがインクルードされています。と、循環参照をしています。このため、コンパイルを通すことが出来ません。

では、どうすればいいか？

答えとしては、クラスヘッダーに、他のクラス参照を記述し、ClassAのコンストラクタでは、自分自身のアドレスを渡しながらClassBをインスタンスして、返り値のアドレスを保持して、以降それを参照する。ClassBのコンストラクタでは、受け取ったClassAのアドレスを保持して、以降それを参照するようにする。以上が、正解となる。

さて、書いてみようか(°Д°)

さて、答え合わせの時間だ。

```
#ifndef _CLASS_A_H_
#define _CLASS_A_H_

class ClassB;

class ClassA {
private:
    ClassB* m_ClassB;
public:
    ClassA();
    ~ClassA();
};

#endif // _CLASS_A_H_
```

```
#ifndef _CLASS_B_H_
#define _CLASS_B_H_

class ClassA;
class ClassB {
private:
    ClassA* m_ClassA;
public:
    ClassB(ClassA* clsA);
    ~ClassB();
};

#endif // _CLASS_B_H_
```

```
#include "stdafx.h"
#include "classA.h"
#include "classB.h"
using namespace std;

ClassA::ClassA()
{
    m_ClassB = new ClassB(this);
}

ClassA::~ClassA()
{
}

}
```

```
#include "stdafx.h"
#include "classA.h"
#include "classB.h"
using namespace std;

ClassB::ClassB(ClassA* clsA)
{
    m_ClassA = clsA;
}

ClassB::~ClassB()
{
}

}
```

様々なオーバーロードの利用方法

その①引数付きコンストラクタ

さて、前にやったコンストラクタは覚えているかな？

インスタンス時に自動実行されるアレだ。

で、インスタンスする際に、例えば、敵キャラクターをインスタンスするとして、その際の初期化で何か値を渡したいとする。

その場合、インスタンスしてから、init()関数を改めて呼び出すとかは非効率だ。

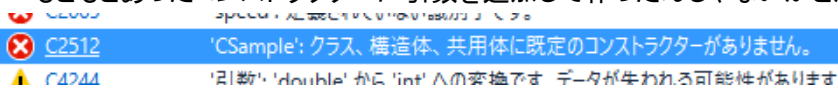
インスタンス時に値を渡して、内部でコールしてしましてほしいよね。

そういった場合に、対応するために、コンストラクタにもオーバーロードを対応することは可能

Csampleクラスのm_numの初期値を0でなく、入力した値を初期値にしてみよう。

もし、コンパイルして、下記のようなエラーが出たら、おそらくは、

もともとあったコンストラクタに引数を追加して作ったんじゃないかと思う。



注意して欲しいのは、引数のないコンストラクタはデフォルトコンストラクタといって、自動で呼ばれるという点だ。

なので、それは標準装備されていないといけないんだ。

ということで、もし、このエラーが出た人は、デフォルトコンストラクタを追加してみよう。

なお、コードはこのような感じだ。

```
9  CSample::CSample(int initNum)
10  {
11      std::cout << "親をメモリーに追加" << std::endl;
12      m_num = initNum;
13  }
```

main側

```
13  int num;
14  CSample *obj1 = NULL;
15  cout << "初期値を入力してください" << endl;
16  cin >> num;
17  obj1 = new CSample(num);           // CSampleをインスタンス
18  obj1->CSample::view();
```

その②便利な継承方法

さて、このようにオーバーロードは自作したメンバー関数ではないコンストラクタにも適応することが可能だ。

で、このオーバーロードの考え方は、何もメンバー関数だけじゃなく、クラス自体にも適応することができるんだ。
実際に書いてみよう。

基底クラスとして、Enemyを作ろう。

それを動的にインスタンスしてみたい。

なお、Updateのメンバー関数を追加し、次のようにしておくこと。

```
10 void Enemy::Update(void)
11 {
12     std::cout << "Enemyの行動処理" << std::endl;
13 }
```

実行すると、上記の文字が表示されると思う。

では、次に、それを継承したDragonとSlimeを作ってみよう。

で、同じく、Update()を用紙、それぞれの敵行動処理のメッセージが出るようにしてほしい。

```
13 int num;
14 Dragon *dragon;
15 Slime *slime;
16 dragon = new Dragon;
17 slime = new Slime;
18 dragon->Update();
19 slime->Update();
20 delete dragon;
21 delete slime;
22 cin >> num;
23
```

多少名前が違うかもしれないが、このように書いたはずだ。

で、例えばゲームでクラスごとに変数の名前を変えていたら、100種類の敵がいたら、100個Update()を呼び出しを書くのかというと無駄だよね。
じゃあ、どうするかといえば、配列にしておいてfor文で回したりしたいわけだ。
それなら、for文を入れても数行で済む。

じゃあ、様々なクラスを一つの配列に入れてしまおう。

どうするかというと、次のように書くことができる。

```
13 int num;
14 Enemy *enemy[5];
15 enemy[0] = new Dragon;
16 enemy[1] = new Slime;
17 enemy[0]->Update();
18 enemy[1]->Update();
19 delete enemy[0];
20 delete enemy[1];
21 cin >> num;
22
```

実行すると動いたね。
継承元の型を配列の際の型として使い、
インスタンス時に実際に使いたいクラスの型名を指定することが可能なんだ。
注意点としては、配列側の型は基底クラスで、
インスタンス時に指定する型は、継承したクラスという点だ。
これさえ、守れば問題はない。

といたいところだが、実行結果を見てみよう。

Enemyの行動処理
Enemyの行動処理

と、基底クラスのUpdate()が呼ばれているっぽい挙動だ。
DragonやSlimeのUpdate()は呼ばれていない。

これだと、困るよね、というか意味がない。
じゃあ、どうするかというと、次のように書くことで回避できる。

```
2  class Enemy
3  {
4      public:
5          Enemy ();
6          virtual void Update(void);
7          ~Enemy ();
8  };
```

基底クラスのUpdate()をVirtual指定するんだ。
Virtualがついていると、継承した場合、基底クラスのものが呼ばれずに、
サブクラス側のものが呼ばれるようになる。
むろん、基底クラスそのものをインスタンスした場合は、virtualがついていても問題なく呼ばれる。
ポリモーフィズムを使った継承をする場合は、基底クラス側のメンバー関数にvirtualを付けて、
使わない場合は、つけなくてもいい。
むしろつけない方がよいといった感じだ。
こういった関数を仮想関数という。
以前、デストラクタにvirtualをつけてもつけなくても現状は、
いいという話があったが、これも同様だ。
ポリモーフィズムを使った継承をする場合は、基底クラスのデストラクタにvirtualを付けないと、
サブクラス側のデストラクタが呼ばれなくなってしまうので、その場合はつけよう。

で、実行すると、

Dragonの行動処理
Slimeの行動処理

ちゃんとそれぞれの敵の行動が呼ばれるようになったね。
こうなれば、Update()の呼び出しは、for文で回せるので、かなり便利になる。

さて、仮想関数をやったので、ついでに純粋仮想関数もやっておこう。
何が違うかというと、仮想関数は基底クラスにその実態があるが、
純粋仮想関数は、基底クラスにはその実態が無い場合の事を指す。
実体がないと怒られるじゃん…と思ったあなた。
実際にやってみよう。
EnemyクラスのUpdate()の実体を削除しよう。

✖ LNK1120

1 件の未解決の外部参照

✖ LNK2001

外部シンボル "public: virtual void __thiscall Enemy::Update(void)" (?Update@Enemy@@@UAEXXZ)" は未解決です。

うん、怒られたね、おけーおけー問題ない。
本題はこっからだ。
では、次に、ヘッダー側を次のようにしてみよう。

```
6 | virtual void Update(void) = 0;
```

これでコンパイルは通るはずだ。
アドレスが0の関数ようなNULLってことで存在しない。
そういった指定ができるんだ。
注意点としては、基底クラスに存在する純粋仮想関数は、
継承する際に必ず作成する必要があるということだ。

基底クラス側に必要ない場合、いちいち実態を書くのが面倒だと思うので、
そういった場合に使う。

オペレータ(演算子)のオーバーロード

最後に、これをおきたいけども、かなり混乱すると思う。
ポリモーフィズムでオーバーロードをやったけども、このオーバーロードという考えは、
自分で作ったメンバー関数以外にも反映できるんだ。
例えば、各種演算子やキャストなどなどにも反映できる。
具体的に言うと、下記の通りだ。

- 1 ()、関数呼び出し演算子、関数オブジェクト
- 2 タイプ()、型変換キャスト演算子
- 3 ->、クラスへのポインタからメンバにアクセスする演算子
- 4 [], 配列の添え字参照演算子
- 5 ++ 後置きインクリメント、-- 後置きデクリメント
- 6 new と delete、メモリの確保と開放

と、まあ、こんなところだ。
今回は紹介として、関数呼び出しの1を紹介しよう。
他については、自分で調べてみてほしい。

では、さっそくやってみよう。
関数を呼び出すときは、どのように呼び出しているかな？
関数名()のようにして呼び出しているよね。
で、オペレーターのオーバーロード時は、
クラスをインスタンスした時の名前に()を付けてあげて呼び出すんだ。
例えば、ClassA obj1;だとしたら、
obj1(付加した引数)
ClassA *obj1のように動的にインスタンスした場合は、
(*obj1)(付加した引数)
のようになる。

引数付きのコンストラクタと見分けが一見つかないが、引数の種類や数の違いで、
見分けるしかない。
なお、言語仕様のには、多様性により許容されている。

実際に書いてみよう。
呼び出し側は、例えばこんな感じだ。

```
16 posSt operator() (double ang) {  
17     double rad = (ang*3.141592) / 180;  
18     return{ cos(rad) * speed, sin(rad) * speed };  
19 }  
20
```

普通に演算結果を格納しようとする、X,Yそれぞれを書かないといけないが、
operatorオーバーロードを使えば、1行で済むようになる。
その分、コードの可視性は下がるので、使いどころには注意しよう。