C++

この授業ではC++を教えます。あ、大先生たちは聞かなくていい程度の話しかしませんので、課題の確認か、DX11の予習でもしておいてください。

目次

C言語おさらい……3
ファイル入出力……3
C/C++言語のポインタの話……4
ポインタの説明はむっつかしー……7
ポインタのポインタアッー……9
C/C++の構造体……11
まめちしき～……14
C++の初歩です……16
std::って書くのが面倒ならusingを使おう……17
16進数を表示したい時はどうするの？……17
はい、ちょっと簡単なのいきましょうか……18
文字列を扱おう……19
C++のクラス……21
とにかくクラスを作ってみる……21
継承……22
ポリモーフィズム……23
仮想関数……25
純粋仮想関数……26
クラスとメモリ……27
確保時にメモリ食いつぶすのはメンバ変数……27
継承時のメモリ上での状態……27
仮想関数を作った時のメモリの状態……27
C言語にない便利な機能……29
参照型の基本……29
オペレータオーバーロード(演算子オーバーロード)……30
オペレータの定義……30
初期化子について(横道にちょっとだけズレますが)……31
オペレータを使って複製禁止にする……33
豆ちしきー……36
#pragma once……36
#pragma region～#pragma endregion……36
テンプレート……37
概説……37
関数テンプレート……38
クラステンプレート……39
配列オブジェクトをつくろー……40
スマートポインタ(初歩)……43
スコープドポインタつくろー……43

簡単なリファレンスカウンタ(非スマートな)ポインタつくろー……47
thisは自分自身を指す……48
自分を参照している人の数を覚えておこう……48
テンプレ化……50
シェアードポインタの作り方……51
例外処理……53
try～catch構文……53
throw……57
RTTI(実行時型情報)……59
typeidについて……60
いろんなキャスト……62
static_cast……62
const_cast……64
dynamic_cast……65
reinterpret_cast……67
メンバ関数ポインタ……68
関数ポインタおさらい……68
メンバ関数のポインタ……71
スレッドプログラミング……73
Windowsの非同期ファイル操作を知ろう……73
初歩(beginthreadとか)……73
ワーカースレッドつくろー……73
スレッドプールつくろー……73
コラムです……73
課題:データ構造とアルゴリズム基礎……74
課題1:リスト構造……74
解説:リスト構造……76
最終課題……83
その1……83
その2……84
その3……84

C言語おさらい

教えます…が、君たちそもそもC言語忘れてるだろ？
いや…全員とは言わないけど、結構怪しくないかな？…。

ファイル入出力

一番怪しいのはファイル入出力…そうだろ？うんうん、今回の課題でもね、できてねー奴が多いもんね。
え？できる…だと？
う～ん、そうだな。悪かった…見くびってた。うん、ゲームに関係していないと面白く無いだろうから…そうだな、ランキングの保存プログラムと復元プログラムを作ってもらおうか。
ランキングってわかるかな？

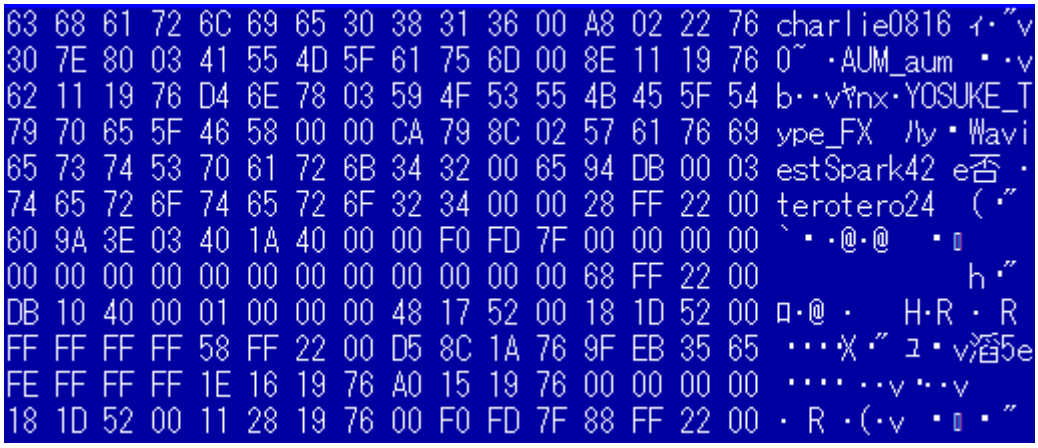
こういう画面ね。最近のゲームでは筐体の電源を消してもランキング情報が保持されることが多いです。つまり内部のハードディスクにランキング情報を保存しているのです。

ああ、仕様か…うーん。そうだなこれにほぼ合わせるかな。名前最大数が16文字、スコア最大が99999999となるようにすること。
まずは肩慣らしだ。
サーバにあるranking_sample.txtをダウンロードして下さい。

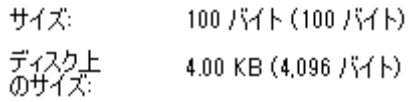
課題その1:

C/C++言語(C#は不可)で、ranking_sample.txt を読み込み、内容をコマンドプロンプトに表示して下さい。この程度ができない人は、ゲーム業界どころか「プログラマ」という職業につくことすら難しいと思いますので、アシカラズ。

課題その2:
1の内容を、バイナリデータとして、saved_ranking.bin に保存して下さい。その時のファイルサイズが 100 バイト以内になるようにして下さい。バイナリデータってのはこういうやつです。



で、サイズってのは、ファイルのプロパティで



ってなってる上の方ね。ハードディスクはセクタで管理しているので、100 バイトだけって保存はできないんだよね。ということで、実際のディスク上では 4KB 食ってるってこと。正しくは上のバイト数を見て下さい。

C 言語として、今回使うであろうコードは printf,fopen,fwrite,fread,fclose,getchar とかではないでしょうか？これが最大限のヒントです。今日はこんなもんです。

これが終わっちゃった人は課題だか DX11 をやっておいてください。

C/C++言語のポインタの話

はい、C#やりすぎでポインタがわからない人が続出してるっぽいんで、ちょっとポインタの話をしておきましょうか。

もう何年も基本情報だのプログラミングだの勉強してきたでしょうから、パソコンの中身についてはなんとなくイメージできてきていると思いますが、パソコンの中身にはメモリっちゅーやつがあります。

まー、このメモリにも色々と種類はあるんだけど、その違いにはもう触れません。もうね、ここまで来てその違いの話しても意味ないし。

ともかく変数を宣言するたびに、パソコンに搭載されているメモリが消費されるわけ。消費と言っても使ったらなくなるわけではなく、使い終わって適切に処理をすれば再利用が可能になるのだ。

で、いろんなアプリケーションがよってたかってメモリを食いつぶしているわけ。なので、自分で作ったアプリケーションが使うメモリは先頭から始まるわけではなく、余っているメモリが、全体の中間からだったりする。予約領域だったりするしね。

で、そのメモリ上のどこか(位置)を一次元的に表すものを「アドレス」と言って、32bit マシンなら 32bit で表されていて、通常は16進で 0xdeadbeaf とかって表現される事が多いわけです。

で、C/C++では、変数のアドレスを返す演算子として&(アンパサンド)を用います。例えば、

```
int a=90;
printf("%d",(unsigned int)&a);
```

とでもしてやれば変数 a のアドレスを表示します。

で、このアドレスを格納する変数の事をポインタと言います。それだけの話です。通常の変数に「型」があるように、このポインタにも「型」情報があります。

実際の中身は、0xdeadbeaf 的な 32bit(or 64bit)共通なんだけど、アドレスから元の値に戻したい時があるよね？その時のためにポインタも「単なるアドレス格納器」ではなくて、器に「int 型だよー、char 型だよー、ユーザー定義型だよー」って書いてある。それが「ポインタ」なわけ。

たまに void*ってのがあるけど、それは器に何も書いてないんだけど、アドレスだけは指し示しているわけ。こういうのを扱う時は、必ずサイズを何処かで明記しなければならなくなる事になります。

で、どういう時にこの void*が出てくるかというと、ライブラリ等でメモリブロックを扱う際に、「この関数は型を問いませんよ」って明記する意味で出てくる。

基本的なところでは memset や memcpy やね。

http://www9.plala.or.jp/sgwr-t/lib/memset.html

http://www9.plala.or.jp/sgwr-t/lib/memcpy.html

DirectX のサンプルでよくある ZeroMemory は結局のところ memset なんやで。ていうかあんなもん使うな。なんで MS がアレを頑なにサンプルに入れるのかわからん。

さて、ポインタを使う理由としては

①関数内で値を書き換えてしまいたい時

②連続メモリにアクセスしたい時

③構造体がやたら大きい時

などが代表的な理由です。③の理由がヨクワカラナイ人もいるかもしれませんが、例えば

BITMAPINFOHEADER を見てみようか？

http://msdn.microsoft.com/ja-jp/library/cc352308.aspx

こんなやつ。

sizeof してみればわかるけど、40 バイトなのだ。ほとんどの人にとってはドゥでもいいと思うんだが、ゲームプログラマからすると 40 バイトは見逃せないデカさなのだ。まぁ、そんな頻繁にやりとりしないなら気にもしないんだが、例えば 100000 個のビットマップを処理するとなればこのコストはバカにならない。

とりあえず、この 40 バイト覚えとけよ。例えば、ビットマップ情報を出力する関数を作ったとする。

```
void OutputBitmapInfoHeader(BITMAPINFOHEADER bih){
    ビットマップ情報出力コード
}
```

はい、そして呼び出し側

```
BITMAPINFOHEADER bih;
～ビットマップをロードし、BITMAPINFOHEADER に情報が格納されている状態～
OutputBitmapInfoHeader(bih);
```

こうすると、どうなるだろうか？関数呼び出しの際に…いや、関数の引数ってメモリ上はどうなってると思う？

関数内変数と同様に、最初にスタックに確保されてるわけ。まぁ、スタックわからんのも多いだろうか、関数用のメモリだと思っとけばいい。

だから、引数を変数として見れない病気の人もいるみたいだけど、こいつ単なる変数だからね。

要はさっきの関数の例で言うと

```
void OutputBitmapInfoHeader(){
    BITMAPINFOHEADER bih;
```

}

メモリ的にはこういうことなわけ。で、こいつが引数になると、呼び出し側の情報が、この変数にコピーされるわけ。だから関数内でいくら値を書き換えても、呼び出し側の値は変わらないわけ。何しろコピーなんだから。

ということは40バイトのコピーが発生する。

メモリ的には

memcpy(&bih,コピー元,40);

ってやってるのと変わらんわけ。

ところが、これをポインタ渡しにしてやればアドレス値を渡すだけなので、アドレスはさっき言ったように、32bitだったりするわけ、32bitって何バイト？4バイトやね。

つまり、40バイトと4バイト…。その差36バイトっ…!圧倒的コストっ…!

とかいう場合に使うわけ。

あと、まだ言いたいことはある。構造体について分かってない子も多過ぎるので書いておこうと思う。

ポインタの説明はむっつかしー

難しいのよ…これが。

「プログラマは2つに分けられる。それはポインタを理解している人間と、理解していない人間とにだ」友蔵…心の俳句

一旦理解すると、理解していない人間の気持ちが分からなくなるし、やっぱり分からない人は分からない事が多い。

もうこれは悟りを開いた人間とそうでない人間くらい違う。ロリコンである人間とそうでない人間くらい違う。

つまり、分かり合えない。

ポインタ教えるときの4段階

①教えてスグ分かる奴

②メタファー(たとえ)で分かる奴

③イメージ(図、表)で分かる奴

④どぎゃんもこぎゃんもなく分からん奴

まぁ、④は…まぁ分かるまで自分で勉強せぇ、分からんもんはしゃーない。今までも数学だの英語だの分からんのあったやろ？しゃーないねん、でもな、やりたいことがあるんやったらしゃーないで済まされへんねん。

やるしかないねん。やらへん奴は仕方ないね、少なくともC系のプログラマには向いていません。①はもはや教える必要なしだからいいです。

教えが必要なのは②からですからね。たぶん、この手のタイプはチンプンカンプンって感じじゃなくて、なんとなく分かってるんだけどーってタイプかな。利用はできてるけど人に説明できないタイプやね。

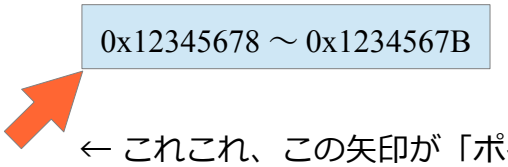
ポインタってのは日本語に訳すると、指し棒みたいな意味なのね。

http://eow.alc.co.jp/search?q=pointer&ref=sa

ほら、マウスポインタっていうでしょッッッ!?ああいう意味。実際、ポインタってのはメモリ上に確保された何らかの情報を指し示すモノなわけよ。

int a=5;

とかであっても、実際にはコンピュータのメモリの 0x12345678 番地やらに配置されているわけやん？



← これこれ、この矢印が「ポインタ」なわけ

そして上の例のように int 型なら 4 バイト食いつぶしてるわけじゃん？そしたらポインタはどれくらい食いつぶしてるのかも知ってる必要があるわけ。

更に言うとポインタを利用するためには元の型を知っておかなければ使えねーじゃん？なのでポインタは元の型を覚えている作りになってるわけ。だからポインタの宣言は

int* p;

みたいに、型名の後にアスタリスクがついていたりするわけ。要は後で利用しやすいように「これは int 型を指し示すポインタですよ」って強調してるわけ。

もしそんな必要がなく、ただただアドレスだけを指し示したいのなら

void* p;

でいいわけだし。ね？ポインタの宣言が"型名*"の形式になっている理由とか意識したことある？別になくてもいいけど、なかなか理解できない人は、この理由を意識しておくといいでしょう。

以上のことから、例えば

int* a=0x12345678;

char* b=0x12345678;

といったように、全く同じアドレスを指し示している2つのポインタ、これは意味が違う。ぜんぜん違う。ということを理屈的にも直感的にも理解していただきたい。何がちゃうかというと

だから*a と*b は同じアドレスですが、全く違う結果になることを知っておきましょう。

また、

++a;

++b;

この時のポインタの進み具合も変わります。

aはint型のポインタなので、++aで4バイト進み、++bはchar型なので1バイトしか進みません。つまり結果的には

aは0x1234567Bになり、

bは0x12345679になります。

分からん奴はわかるまで勉強しましょう。C/C++プログラマになりたければね。

で、ここできちんと認識しておいて欲しいのは、ポインタが指し示しているものはアドレスであるので、アドレス自体は32bitの場合は4バイトのunsigned intと等価な型になっているわけよ。

とにかく数値で表されているわけ。

そうすると、当たり前の話なんだけどポインタを宣言した時点でまたメモリ上に4バイトは確保されているってことなのね。

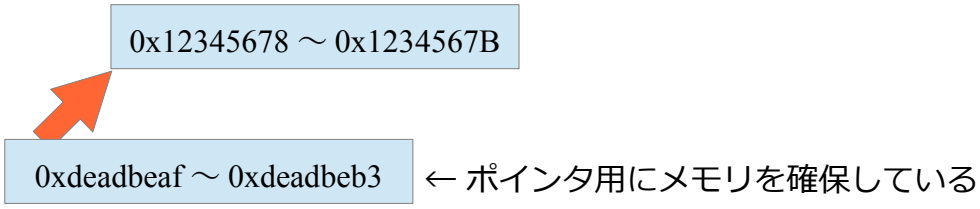
だから例えば

int a;

int*b=&a;

なんてやって

int型の変数aが0x12345678だったとして、ポインタbがaのアドレスを指し示しているとすると、例えば、b自体のアドレスは0xdeadbeafであり、そっから4バイト確保されることになる。



ポインタのポインタアッー

ポインタの理解できていると思っている人でもたまに

「ポインタのポインタがわからん」という。

これはね、はっきり言っちゃうと理解してへんねん。え？なんでかって？

ポインタもまた変数であり、そいつも変数である以上はどこかしらにメモリが確保されているという重大な事実がわかってへんねん。

つまり、やっぱりメモリとかアドレスとかポインタが理解できていない。もうひと頑張り

が足りていないわけ。

例えば、アドレスがこういうものであるとする。この例では 0x12345678 とかって書くとヤヤコシイのでメモリ上に 0～100 までアドレスが並んでいるとする。

メモリは一次元なので、一行目は 0～9 番目、二行目は 10～19 番目が並んでいて後は同じようにって感じだと思ってくれ。

さて、まず例えば int a=5;とかって書いたとする。それが実行されるとアドレス上の何処かに 5 が書き込まれるわけや。自分でも適当に配置してちょ？

で、下の図だと、そのアドレス値は 15 であることがわかる。アドレスであることを分かりやすくするために16進数で書いとくと 0x0F ね。

ここでこの 0x0F というアドレス値を何かの変数に保存したとする。

int* b=&a;

そうするとこれもどこかのアドレスに確保されているわけや。つまり b にもアドレスがあると…。今回の例だと 63 にある。16 進にすると 0x3F。

これも変数に入れるならどっかに入っていることになる。

int*c=&b;

NULL									
					a=5				
							c=0x3F		
			b=0x0F						

で、ポインタのポインタの話だが、ポインタのポインタを扱う時の主役はアドレス値なのね。今回で言うと b のこと。

DirectX11 の Create 系の関数とかでよくある、あの最後の引数にポインタのポインタを渡すってのは

ポインタ型を b とすると

Create なんとか(1,2,3,&b);

ってなるわけ。この例であれば最後の部分に b のアドレス(この例では 0x0F)が入っている。で、例えば Create なんとかの場合は b=NULL だったりするわけ。

うん、でそのアドレスを渡すということは、Create なんとか関数内で適当にメモリを確保して作られたオブジェクトのアドレスを変数 b に放り込めるってことなんだ。

わかりにくいかな？

b=NULLにしていると仮定する。

で、下のような関数がある。最後の引数がポイントなのね。外側からポインタ型の変数のアドレスが渡される…。関数内でこのアドレスの指す内容を書き換える。Createの場合であれば大抵は内部でメモリが確保され、確保したアドレスを以下のように値として渡してやる。

```
void Createなんとか(int a, int b , int c, int** d){
    *d=new int;
}
```

newは渡された型に応じて適当にメモリを確保し、そのアドレスを返すものだ。そのアドレスが、ポインタのポインタで渡されたdに入っている。いや、*dなので、アドレスdが指し示す先(ここもアドレス)に、新しく確保したメモリアドレスを渡しているのだ。

結果として

```
int* q=NULL;
Createなんとか(1,2,3,&q);
```

とやれば、本来NULLのアドレスに、きちんと確保されたアドレスが入って来るってことです。

C/C++の構造体

もちろん構造体自体は作れるだろう。作れない人のお帰りはあちらです。

ただ、その扱いがよーわかってない奴がいる。プロでもっ…!馬鹿なっ……!

例えばさっきのBITMAPINFOHEADERで説明する。

```
BITMAPINFOHEADER a;
BITMAPINFOHEADER b;
```

と2つのBITMAPINFOHEADERがあるとする。なんかしらの理由で、変数aの内容を変数bにコピーしたいとする。

中にはこんな奴がいる(アッー!って思った人、別に気にするな。これから直せばいいんだ)。

```
memcpy(b,a,40);//aの内容をbにコピー
```

間違ってはない、間違ってはないんだが…これはアカン。これだとわかってないと言わざるを得ない。

そもそも、memcpyのサイズに手書きで40って入れてるのも気に食わん。

なんかの拍子に41になった瞬間に破綻するってことがわからんか？クラッシュしまっせ？

じゃあ、こうか？

memcpy(b,a,sizeof(a));//aの内容をbにコピー

ん～、だいぶ良くなったけどぉ…もうひと頑張りねぇ。

それでは正解

b=a;//aの内容をbにコピー

これが正解。え？わかってるって？馬鹿にすんなって？そうなん？結構memcpy使ってるコードも見るんだけどね。すまんすまん。

アレだろ、どっかのサンプルコードを盲目的にコピーしてっからそんな風になるんだわー。っべーわー。サンプル盲信怖いわー。マジっべーわー。

フツーに、構造体であっても変数なんだから、intとかcharとかfloatの変数と同様にイコールでコピーするわけ。

はい、なんで無駄にmemcpyやる人が増えたのか、僕には予想がつく。これがうまくいかないから、なんか勘違いしているのだ。

```
int a[10];
int b[10];
b=a;
```

ああ、そらうまくいかんわな。配列の名前はアドレス示すんで、これじゃあうまくいかんから確かにmemcpy使うしか無いわな。

ないかな？

ちょっとセコ技使えばいルル。

```
struct Array{
    int values[10];
};
```

とでもしてやって

Array a;
Array b;
b=a;
とでもしてやれば、構造体としてコピーされる。だが、もちろんこういうのはダメだ。

struct Array{
	int* values;
};
Array a;
Array b;
a.values=new int(10);
b.values=new int(10);
b=a;//これではポインタしかコピーされない。

あー、馬鹿にすんなってか。すまんすまん。ちょっと最近授業してると不安になってきてね…

んじゃさ、構造体を0で初期化する時って、どうやってる？

ZeroMemory(&a,sizeof(a));

とかやってねーだろうな？ええー ZeroMemory～？ZeroMemory が許されるのは小学生までだよね～!!キモーイ!!

だからといって memset(&a,0,sizeof(a))も使うんじゃねーぞ。これの何がダセーってあなたサイズ間違えると飛ぶってやつよ。

はい。正解です。
BITMAPINFOHEADER a={};
BITMAPINFOHEADER a={0};
BITMAPINFOHEADER a=BITMAPINFOHEADER();

です。好きなのを使ってくれ。最後のやつの何が強いってあなた上のふたつは宣言初期化時にしか使えへんけど、最後のやつはどこでも使えるって点よ。

どういうことかというと、BITMAPINFOHEADER()って書いた時点で、右辺側に0で埋められたオブジェクトができるので、それを左辺に代入していると考える。つまり色々とこのaに変更を加えた上で、初期化しておきたい場合はa=BITMAPINFOHEADER();としてやりゃいい。

```
BITMAPINFOHEADER a={};
a.biSize=99;//biSizeが99に変更されてしまった!!
a=BITMAPINFOHEADER();//でもオールOになる。
```

というわけ。

ちなみに、ここで罠を仕掛けていますが、BITMAPINFOHEADERはBITMAPFILEHEADERの後に来てますんでね。イキナリBITMAPINFOHEADERをロードしようとすると変なデータが入ってきますよ。

対処は簡単で、最初にfreadでBITMAPFILEHEADERぶんロードしておけば、ファイルカーソルが先に進むため、その後でBITMAPINFOHEADERをロードすればいい。

まめちしき～

日常に使えないムダ知識をあなたに…

ねぇ知ってる？C/C++言語の配列って、配列名とインデックスを入れ替えても正常に動作するんだよ？わからない!?

たとえば、

```
int a(100);
```

って宣言して、a(54)って書いたら、aっていう配列の55番目の要素にアクセスする。これはわかるよね？

じゃあ、54(a)なんて書ける…と思う？

それが書けるんだわー。マジで。ちなみに

a[54]==54[a]

でございます。

だから例えば、

```
int main(){
	int a[10]={1,2,3,4,5,6,7,8,9,10};
	printf("%d¥n",a[5]);
	printf("%d¥n",6[a]);
}
```

とでも書くと、画面上に67と出力される。

これはね、脳味噌がプログラマ脳になりきってて、メモリ的に頭を働かせることができたなら理解できるんだけど、まだまだそこまでじゃない人も多いと思うので……人間を辞める覚悟があるなら、こんなのはスグにわかるようになります。

要は、配列の

配列名[インデックス]

って形式は、メモリ上の特定の場所の要素を表すシンタックスシュガー(糖衣構文)でしかないってこと。

ほら、C言語の場合、配列名ってポインタ渡しとして扱えるでしょ？つまり、配列名の中身には

配列名==0xdeadbeaf

ッて感じでアドレスが入っているわけ。

で、

[インデックス]

の部分は、コンパイラはどういう風に扱うかというと、その配列名のアドレスからのオフセットを表しているわけ、だからC言語ってのは最初の要素が0なんだ。つまり数式で書くと

(配列名のアドレス＋インデックス)←この場所にある値を返す

っていうことになっている。つまりこれならば別に

(インデックス＋配列名のアドレス)←この場所にある値を返す

でも構わないってこと。

だから

a(b)だろうがb(a)だろうが構わないわけ。まめちしき①おわり。

C++の初歩です

ここからは、C言語で書いたであろうコードを少しずつC++らしくしていきましょうか。
Cでは標準出力をprintfを使って書いていたが、
C++では、もちろんprintf使えるんだが、C++らしくするにはcoutを使用する。
こいつを使うには

```
#include<iostream>
```

をインクルードしておく。
で、cout等は、C++の標準ライブラリの一種で、stdというnamespaceで括られているため

```
std::cout<<"文字列"<<std::endl;
```

と書かなければならない。coutやendlの前に、std::ってつけるんだ。なお、endlは改行って意味です。

で、こいつ、何が便利なのかというと、

```
std::cout<<"文字列="<<数値<<std::endl;
```

ってやれば別に%dなんて使わなくても、数値を出力できる点です。

std::って書くのが面倒ならusingを使おう

しばらくC#いじってた人は、usingを何度か見たことがあると思うが、基本的にはアレと同じです。C#の場合、usingを使わないとやたらとピリオドを打つ羽目になります。それと同じような感じでここで、

```
using namespace std;
```

と書いておくとstd::を書かなくてすみます。

はい課題3です。

サーバにsample.bmp置いてますので、このcoutを使用して、ビットマップの幅、高さ、最大色数を出力して下さい。

16進数を表示したい時はどうするの？

はい、C言語のprintfの場合printf("%x",255);なんてやればffと表示されていたはずです。C++のこのcoutを使った場合、%dだの%xだのの指定はデキマセン。さて、それではどうしましょうか？というところで出てくるのが、std::hexです。こいつはマニピュレータという機能の基本的なもので、出力の書式を変更する機能の一つです。どうしてhexなのかというと英語で16進数のことを

```
hex(hexadecimal)number
```

というからです。本当はhexadecimalnumberっていうんですが、あんまり長いのは向こうさんも嫌いなので、略してるわけです。色々言葉を略するのは日本だけの文化じゃないんですね。

で、使い方はどうするのかというと、

```
unsigned int a=1234567890;
std::cout<<std::hex<<a<<std::endl;
```

のように、出力したい値の前にhexを置いてあげます。そうすると、それ以降に書いた値が16進数で表現されます。

まぁ、ほとんど使わないですけどね。

sprintfみたいな関数はあるのん？

はい、ちょっと簡単なのいきましょうか

久々なのでちょっと簡単なのにしましょうか

そろそろ次のあいさつ当番(10/7(月))も近づいてきましたね。そうですね。そうなのですよ。はい、ではランダムで5人選び、その名前を出力するプログラムを書いて下さい。
名簿はテキストで shuseki.txt においています。

※ちなみに、今回の 10/7 のあいさつ当番は私服でいいですよーやったー!!

これをファイル読み込みして、ランダムで5人の名前を出力して下さい。なお、既に押川、新藤、冨安、早田、山口は選出されているため、これを除外して選出するように。

通常の rand 関数は精度が悪いため、「メルセンヌツイスタ」を使って乱数出力して下さい。

メルセンヌツイスタについては自分で調べて下さい。ググる能力も、これからは大事な能力の一つです。

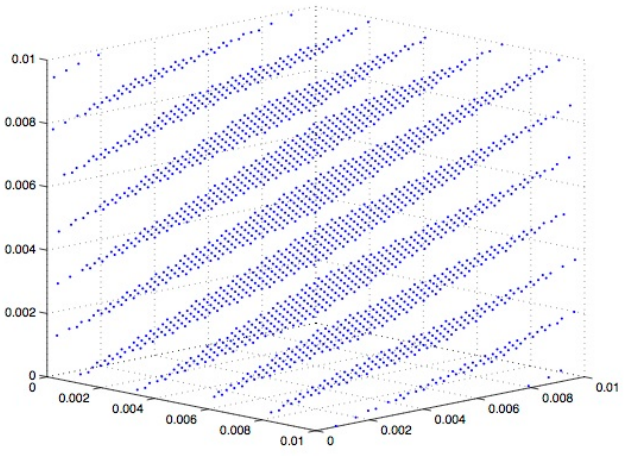
よろしく!あ、公正なプログラムでない(自分は絶対選出されない等)事が判明しだい、優先的に挨拶当番にまわされますので、よろしく。

メルセンヌツイスタについて説明しておくと、これは乱数発生器なんです。rand()と同じようなものではあるんですけど、実を言うと rand()関数ってのは、乱数としての精度が悪いのです。どう悪いのかというと、ものすごーく大きな視点から見ると、パターンができてしまうんだ。パターンが出ている時点で、乱数としては性能が悪いことになる。

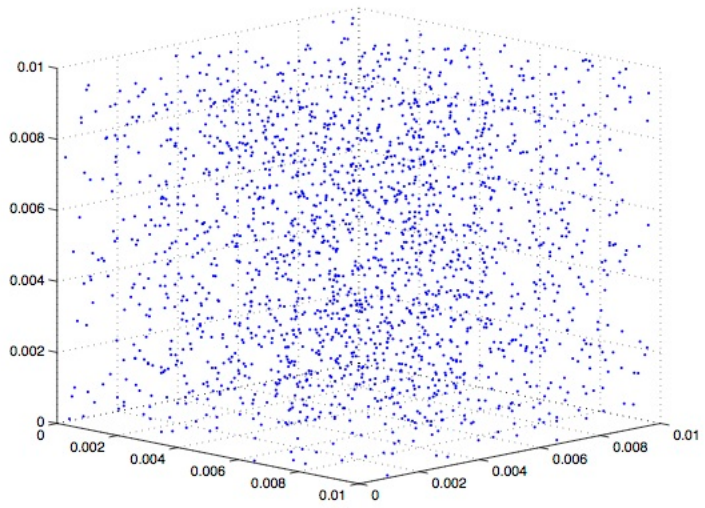
rand()が採用している方法は線形合同法ってやつで

$$X_{n+1} = (A \times X_n + B) \bmod M$$

で与えられるものです。ちょっと出来る人は、この式を見た時点で「アレ?これオレでも10分で作れるんじゃね?」って思いますよねー?
これだと、



分布がこのように一定のパターンになってしまっています。これが☆メルセンヌツイスタ☆だと



図のように、パターンがほぼ見られないバラけた乱数となります。この乱数発生器の何が素晴らしいかというと、こんなに精度が高い割にはコスト(計算時間)がそんなにかからないってことです。

文字列扱おう

文字列です。C/C++言語は文字列を扱うのが面倒だと言われます。そうなんです。C/C++はバイナリを扱うには素晴らしい言語なんですけど、こと文字列を扱うとなると非常にヨワヨワなのです。

言ってしまえば熱血硬派な言語なんです。というか、その昔は文字列にそこまでの要求がかかってなかったとも言えるんです…。

とはいえ、昨今ではC/C++言語においても文字列を使う機会は多くなっています。で、ここでC言語よりもC++の方が使いやすかったりします。

例えばC言語では

```
char*s1="abc";
char*s2="def";
```

の2つの文字列をつなげて新しい文字列を作りたい場合には…何を使うんだったかな？

そう、strcat関数でしたね？strcat(文字列1,文字列2)で、文字列1に文字列2が連結されるのでした。ですが、

```
#include<stdio.h>
#include<string.h>
int main(){
	char*s1="abc";
	char*s2="def";
	strcat(s1,s2);
}
```

なんて書くとクラッシュします。

なんでか分かりますか？分からない人はもう一度「独習Ｃ言語」を読み直しましょう。
s1にs2を連結するのですが、実はs1はこう書いた時点で内部的には

```
char s1[4]="abc";
```

と同じことになっているのですね。そりゃクラッシュする。うん。だから予め多めに確保して

```
char s1[16]="abc";
```

とかってしてないといかんわけ。

```
#include<stdio.h>
#include<string.h>
int main(){
	char s1[16]="abc";
	char*s2="def";
	strcat(s1,s2);
	printf("%s",s1);
}
```

はい、誠にイカンですね。もうちょっと柔軟にいかんものかということで、C++にはstringってのがある。これもstdファミリーだ。string.hは今すぐ捨てろ。

そして

```
#include<string>
```

と書こう。hがついていないのがポイント。これがSTLの('A`)人('A`)ナカーマである。こいつはすげえ。
何がスゴイかというと、連結とかすっげぇスマートに書ける。

```
string a="僕と";
```

とでも書いておいて

```
a+="契約して";
a+="魔法少女になってよ";
```

とでも書いてやれば、ドゥンドゥン文字列が増えていく。これってノベルゲーとかに使えそうでしょ？
はい、それでは練習として、自分の学籍番号とスペース1つ分と自分の名前を連結してcoutで出力してみよう。
これから毎日連結しようぜ。

C++のクラス

とにかくクラスを作ってみろ

さぁ、C++といえばクラス。クラスといえばC++です。
まず、クラスを作ってみましょう。C++でのクラスの作り方は

```
class クラス名{
    アクセス修飾子1:
        メンバ変数1;
        メンバ変数2;
            …
        メンバ関数1();
        メンバ関数2();
            …
    アクセス修飾子2:
            …
};
```

と言った具合になります。C#に慣れてしまっちゃった人は最後の;(セミコロン)忘れがちになりますので注意しましょうね。

モチロン、ここでキーになるのはclassってキーワードです。これを構造体の宣言のように最初に記述することで、これはクラス扱いになります。これはC#でも同じだから簡単だよね。
ガッチガチのC言語脳の人は、まずはstructで書いてみて、structの部分をclassに変えてみましょう。それだけで、とりあえずはC++のクラスを作った事にはなります。

で、次に気になるのは「アクセス修飾子」です。これはC#でもやっているでしょうから、何となく分かる人も多いかもしれませんが
public:とprivate:とprotected:という「見せるのか」「見せないのか」「子供にだけ見せるのか」という指定を行うものです。
なお

- **publicはみんなに見せちゃう**
- **privateは誰にも見せない**
- **protectedは自分の子供にだけ見せる**

といった区別が有ります。
C#での記述の場合は、関数の前にこのアクセス修飾子を記述するルールになっていますが、C++の場合は、public:などと書いた下の行にあるメンバ変数、メンバ関数は全てpublic扱いになります。

```
class クラス名{
    private:
        メンバ変数1;
        メンバ変数2;
            …
        メンバ関数1();
        メンバ関数2();
            …
```

private扱い

```
    public:
        …
        …
};
```
public 扱い

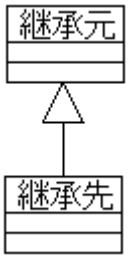
なお、これもC#から来た人は忘れがちですが、**アクセス修飾子の後には:(コロン)を書く必要**がありますので、忘れないようにして下さい。

乱暴な教科書(HPなど)とかだと、「わかんねーならとりあえず全部publicにしとけ」って書いてます。…まぁ、ある意味間違ってはいませんけど、もうそれクラスじゃなくていいじゃん。っていう話になります。
で、ヨクワカラナイかもしれないのが、protected:(自分の子供にだけ見せる)なのかもしれませんが、これは「継承」の話をやった後にすることになります。

余談ですが、C++の場合は構造体(struct)であってもメンバ関数を定義することができます。意外と知られていません。

継承

ええ、早速継承です。C#で継承をやったことのある人もいるかもしれませんが、若干C#とは違うので、変に混同しないようにしましょう。
そもそも継承ってのはなにか、インヘリタンスとも言われるものですが、教科書的に言うと、継承元と継承先とあって、継承先は継承元の性質を受け継ぐというアレです。受け継ぐから継承ね。



よくある図だとこういうものね。継承元がpublicかprotectedで宣言したメンバは、継承先でも使用できる。つまり受け継いでいる。だから継承ってわけ。

さて、継承の記述はどうやるのか
```
class継承先:public継承元{
    (中略)
};
```
と言った具合になります。
ポイントは継承元の手前にあるpublic。C++では継承の際にpublic継承やprotected継承やらprivate継承やらがあります。
あるんですが、大抵の場合はpublic継承を使用しますしC#の継承はpublicですので、基本はpublicをつけておいて下さい。

で、まぁ、"オブジェクト指向言語は再利用性が高い"。教科書的にはこの「継承」をできる事が

理由に挙がっていることが多いと思うが、そうじゃないと思う。いや、確かにこの継承は親の関数を使いまわせるので、確かに「再利用」ではあるし、それは間違っていないのだろう。

だが、この後にお話するポリモーフィズム(多態性)を理解しないならば、再利用性ウンヌンは片手落ちであろうと思う。

ポリモーフィズム

ポリモーフィズム…多態性…どっちにしろわけわからん言葉である。これ、分かってる人間が、分かってない人間に説明すんのはすっげーツライ。ポインタを説明するのに似ている。
だけどとにかくヒトコトで言っておくと

「**異なるクラスを一緒くたに扱えるようにする機能**(仕様とも言えるし概念とも言える)」

である。
一緒くたに扱えると何がウレシイかというと、異なるクラスAとクラスBがあったとして、どちらにも同じ命令を出せる。

でもクラスAとクラスBは異なるため、挙動が異なる。同じ挙動でもいいけど、それだったら同じクラスでいいじゃん？ここを別のクラスで別の挙動ができるからウレシイわけ。
例えば、敵が何種類もいて、パラメータだけで挙動を分けられない時ってありますよね？

こういう場合は例えばUpdate()って関数で敵の動きを定義していたとするとUpdate関数の挙動を変えなきゃいけない。
こういう場合は2つのクラス…例えばSkeltonとSlimeを用意し、それぞれにUpdate関数を作って、それを実行する必要がある。

で、普通にSkeltonとSlimeが別のクラスなら

```
Skelton* skelton=new Skelton();
Slime* slime=new Slime();
```

とか書かざるを得ない。だが、スケルトンやらスライムは一匹ずつ存在するわけではない。そいつらに行動させようとすれば

```
Skelton* skeltons[100];
Slime* slimes[100];
```

とでも書く必要がある。で、行動させようと思ったらイチイチ

```
for(int i=0;i<100;++i){
    slimes[i]->Update();
    skelton[i]->Update();
```

```
}
```

と分けなければいけなくなる。これはウザい。どのみち Update 関数を呼んでるだけなのに…同じことをしてるだけなのに。
更に言うと敵の種類が Skelton,Slime,DarkKnight,Evileye,Devil,Cyclopse…と、増えていくととってもやっていられなくなる。

これを「ポリモーフィズム」の考えを使えばスッキリ書くことができる。
ポリモーフィズム、ポリモーフィズムと言うけど、実際に使うのはそれほど難しくありません。
どうするのかというと、特定の**基底クラス**を用意し、「同じように使う」クラスを**基底クラス**から継承させればいい。
つまり、例えば敵の基底クラス Enemy を作ります。

```
///敵基底クラス
class Enemy{
    public:
        void Update(){
            //仮処理
        }
};

class Skelton : public Enemy{
    public:
        void Update(){
            //スケルトンの更新処理
        }
};

class Slime : public Enemy{
    public:
        void Update(){
            //スライムの更新処理
        }
};
```

例えばこのように継承してしまえば、継承先のクラス、この例で言うとスケルトンやスライムは一緒くたに Enemy(敵)として扱えるわけです。
これはすごい話ですよ。

```
Enemy* enemy1=new Skelton();//中身はスケルトンだが「敵」として扱える
Enemy* enemy2=new Slime();//中身はスライムだが「敵」として扱える
```

が許されちゃうわけです。つまり内容の異なる全ての「敵」を敵倉庫(Enemy の配列やベクタ)に放り込んでおくことができるわけです。

```
Enemy* enemies[256];
enemies[0]=new Skelton();
enemies[1]=new Skelton();
enemies[2]=new Slime();
enemies[3]=new Slime();
enemies[4]=new DarkKnight();
enemies[5]=new Evileye();
enemies[6]=new Devil();
enemies[7]=new Cyclops();
    :
```

なんていう風に定義して

```
for(int i=0;i<256;++i){
    enemies[i]->Update();
}
```

で更新の命令を出すことができるわけです。

仮想関数

ところが、そうは上手くいかないんだよね。残念なんだけど。
例えば

```
class Enemy{
    public:
        void Update(){
            cout << "敵基底アップデート" << endl;
        }
};

class Skelton : public Enemy{
    public:
        void Update(){
            cout << "スケルトンアップデート" << endl;
        }
};

class Slime : public Enemy{
    public:
        void Update(){
            cout << "スライムアップデート" << endl;
        }
};
```

とでも定義してやって

```
int main(){
```

```
    Enemy* e1=new Skelton();
    Enemy* e2=new Slime();
    e1->Update();
    e2->Update();
}
```

とか実行してみ？どういう結果が出るのか自分で確認してみ？

はい、"敵基底アップデート"と二回表示されるだけです。

…あれれぇ？どうして違う挙動をしないのよぉ～～～ッ!!

それはな。C++いやC#でもそうなんだが、「型」がEnemyで有る以上は、中身がスケルトンだろうが、スライムだろうがEnemyの関数が呼ばれちゃうんだね。

なんでやねんと思ってしまうのだが、型自体が基底クラスなので仕方がない。そういう仕様なのだ。では、どのようにすれば実体のほうの関数をコールすることができるのか？

というところをなんとかしたければ
C++においては「仮想関数(バーチャルメソッド)」という実装をしなければならない。

仮想関数の実装は至って簡単である。
基底クラスの「派生クラスで挙動を変えたい関数」の頭にvirtualと書くのである。

```
class Enemy{
    public:
        virtual void Update(){
            cout << "敵基底アップデート" << endl;
        }
};
```

ねっ？簡単でしょ？きちんと挙動が変わることを確認していただきたい。

純粋仮想関数

で、どうせ「継承」するんだったら基底クラスの関数をイチイチ定義するのが('A`)ﾏﾝﾄﾞｸｾっていう意見に対応した仕様が**純粋仮想関数**です。
こいつも簡単。もう基底で実装する必要がない関数にvirtualをつけて、最後が面白いんだけど、関数の最後に=0をつける。それだけです。

```
class Enemy{
    public:
        virtual void Update()=0;
};
```

ご覧のように関数の中身を書く必要がなくなります。こういう関数のことを純粋仮想関数と言い、こいつを持っている基底クラスを純粋仮想クラスと言います。

ただし制約があって、**純粋仮想クラス自身は実体を持つことができません**。覚えておきましょう。

クラスとメモリ

C++のクラス型の変数を確保時はメモリ上でどのようになっているのかを考えてみましょう。

確保時にメモリ食いつぶすのはメンバ変数

メモリ的に考えなければイケないのは、メンバ関数ではなくメンバ変数の方です。
つまり

```
class A{
    public:
        int a1;
        int a2;
        void Tekitou(){
            int p=9;
            int b=6;
            cout << p << endl;
        }
};
```

と定義した場合、sizeof(A)は中にあるa1,a2メンバ変数により8バイトとなります。関数の中でいくら確保してようと関係ありません。
ここまではいいでしょうか？

継承時のメモリ上での状態

例えばAから継承してクラスBを作った場合、メモリ上の配置は例えば

```
class B : public A{
    public:
        char b1;
        int b2;
        void QuuBee(){
        }
};
```

であれば

a1
a2
b1
b2

といった配置になります。簡単でしょう？

仮想関数を作った時のメモリの状態

と、ここまでは直感的でしたし、そこまで難しくもない話だったんですが、このへんからはちょっとマニアック。

仮想関数を実装した場合、型的には基底クラスを示しているのに、どうして派生先クラスの関数の方を指すことができるのでしょうか？
誰かが派生先クラスを指してくれているはずですよね？
そう…誰かが指してくれているんですわー。

それはC++プログラマの間ではvptrと呼ばれるもので、『仮想テーブルポインタ』ってやつなのだ。こいつは見えないポインタであり、いじることができないのだが、ちゃっかりメモリだけは食いつぶしてくれる。
例えば

```
class A{
    public:
        int a1;
        int a2;
        void Tekitou(){
            int p=9;
            int b=6;
            cout << p << endl;
        }
};
```

なら前述のとおり8バイトのクラスとなるが、

```
class A{
    public:
        int a1;
        int a2;
        virtual void Tekitou(){
            int p=9;
            int b=6;
            cout << p << endl;
        }
};
```

上記のようにvirtualいっこつけるだけで、4バイト食いつぶすのであるッ!!つまり合計12バイトとなる。この食いつぶしているのが、仮想テーブルポインタ(vptr)である。
このvptrが仮想テーブル(vtable)へのポインタを持つことにより、実行時にどの関数を実行すべきかわかるのである。

じゃあ、仮想テーブルってなナンジャラホイってゆーと、vptrが指定するための関数のポインタがズラーッと並んだテーブル(いわば配列)があるわけ。

各クラスのオブジェクトは生成された時点でコッソリとこのvptrを持っておき、そいつがテーブルを参照→テーブルは実際に実行されるべき関数のポインタがある→実行→(°Д°)ウマー
となる。

ここでわざとぼやかしてるが、vtableのメモリ上でのあり方…なんだが、これ説明するとC++自体を実装する話になりかねないためもう少し先に進んでからにしよう。
今はインスタンス生成時にどこかにvtable用の領域が確保されていると思っておいてく

れたまえ。
　特にゲーム会社を狙っている人に認識しておいてもらいたいのは、C++言語ってのは(C++だけじゃないと思いますけど)、**自分で定義した覚えのない部分プラス見えない部分でちょっとずつちょっとずつメモリを食いつぶしているという事実**だ。

C言語にない便利な機能

参照型の基本

　C++言語には「参照」という機能があります。
　ポインタのようでポインタでない。ちょっとだけポインタっぽい、見た目は普通の変数…とっても不思議なモノであります。
　使い方だけ言っておくと参照ってのは

　宣言はこうします。

　型& 変数=参照先;

　これが参照です。型の後に&(アンパサンド)をつけてあげると、その変数は「参照型」になります。
　ポインタのように右辺のオブジェクトを実体として指し示すようになります。
　どういう事かというと、

```
int b=90;
int& a=b;
```

　と書いてあげて
　a=50;
　とでも書いてあげると、aはbを指し示しているだけなのだから、aを変更するということはbを変更するということになります。

　つまり、上のコードの結果は
　aもbも50となります。

　関数の引数としても同様で

```
void CheckHitEnemy(Player& player,Enemy& enemy){
	if(IsHit(player.Rect(),enemy.Rect()){
		player.Damage();
	}
}
```

　のように宣言しておけば、ポインタを使わずとも&をつけるだけで、playerやenemyそのものに変化をつけることができます。

だからほらよくあるじゃないですか、呼び出し側から渡した変数を関数内で変更してもらいたいがためにわざわざポインタで渡したりするじゃないですかーやだー。

```
void func(int* p){
    *p=10;
}
int q=20;
func(&q);
cout << q << end;//qは10を出力します
```

こんなワケワカなことやらにゃいかん。ポインタなんか使いたくないんじゃ、アドレスのことなんか考えたくないんじゃ。という時に参照を使えばホラこの通り

```
void func(int& p){
    p=10;
}
int q=20;
func(q);
cout << q << end;//qは10を出力します
```

ねっ、簡単でしょ？

オペレータオーバーロード(演算子オーバーロード)

オペレータオーバーロードってのは、何か…それはC＋＋言語の演算子を自分ルールにできるって機能なのだ。

極端な話、足し算を掛け算にしてもいいのだ。そういう無茶苦茶な機能なのだ。ムチャクチャできるのがC++言語なのだ。

演算子ならば全てです。

=,==,+, -, +=, -=, *, /, <<, >>, new, delete, &, *, ->その他もあったりなかったり…します。ちなみにnewとdeleteは「演算子」の仲間です。

オペレータの定義

オペレータオーバーロードは、オペレータ(演算子)の意味合いを定義するもので、**関数(メソッド)**として定義します。

```
戻り値 operator 演算子(パラメータ){
    ほにゃららほにゃらら
}
```

定義はこうです。簡単でげしょ？例えば2次元ベクトルを表現するVector2クラスを自作するとします。そうすると

```
///2次元ベクトルクラス
class Vector2{
```

```
    public :
        int x;
        int y;
        Vector2():x(0),y(0){}
        Vector2(int inx,int iny):x(inx),y(iny){}
};
//ベクトル同士の足し算
Vector2 operator+(const Vector2& rv,const Vector2& lv){
    return Vector2(rv.x+lv.x+rv.y+lv.y);
}
```

こんな感じに定義します。今回は「ベクトル同士の足し算」を+演算子で定義しています。これを使うにはなにか特別な書き方が必要かというと、そんなものはありません。

```
Vector2 a(1,2);
Vector2 b(3,4);
とでも宣言してやって
a+b
```

これで終わりです。今度こそ本当に簡単でしょ？便利でしょ？

初期化子について(横道にちょっとだけズレますが)

ちなみに余談ですが、書いちゃったので解説しておくと、

```
        Vector2():x(0),y(0){}
        Vector2(int inx,int iny):x(inx),y(iny){}
```

これ、この行に疑問を持った人も居るかもしれません。…なにこれ？何やってんの？とこれは**初期化子**といってクラスのメンバを初期化するために使うものです。
書式は

コンストラクタ(パラメータ) **： メンバ1(初期化値1),メンバ2(初期化値2).....**

と言った具合に使用します。コロンを書いた後に自分の持っているメンバ変数を書き、その後、初期化したい値にカッコをつけた状態でコンストラクタを宣言すればいいのです。
「え？そんなもん必要なん？」

確かにそうだ。こんなもん知らなくてもプログラムは組める。だがこの授業は「C++マニアックス」だ。一応教えておく。これを知っておくとSourceForgeのソースコードとかを見るときにイチイチつまらなくても済む。

また、便利な話としては、const型、参照型のメンバ変数を持っている場合は、本来こいつらにはクラスオブジェクト生成後には値を変更できない→このため初期化すらできない。という非常に悲しいことになるのであるが初期化子を使用すれば可能なのである。

例えば…
```
class A{
    public:
        const int nobita;//エラー!!constだったら値を入れろや!!
        A(){}
};
```

怒られた(’ •ω•`)…じゃあ

```
class A{
    public:
        const int nobita=1;//エラー!!こんなところで初期化すんなや!!
        A(){}
};
```

どないせえと(’ •ω•`)ｼｮﾎﾞｰﾝ

こういうところで役に立つのが**初期化子**なのだ。

```
class A{
    public:
        const int nobita;
        A() : nobita(1){}//そしてこの初期化である(`•ω•’ )
};
```

そんなもんです。閑話休題オペレータに戻ろう。

さて､オペレータオーバーロードでは+=も使えるという話を書きました。
+=などの元の値に影響をあたえる場合はクラスメンバとして定義します。
```
class Vector2{
    public:
        int x;
        int y;
        Vector2():x(0),y(0){}
        Vector2(int inx,int iny):x(inx),y(iny){}
        Vector2& operator+=(const Vector2& v){
            x+=v.x;
            y+=v.y;
            return *this;
        }
        :
        (略)
```

って感じで定義します。別に+=した後の値を使用しないのであれば戻り値をvoidにしてても構いません。

オペレータを使って複製禁止にする

さらに、このVector2をコピー不可にしたい場合があるとする(本来Vector2みたいなのは寧ろコピーすべきではあるんだが、まぁ…あくまで例えばの話だ)
クラスとか構造体で作った型ってのは、int型とかのように値をコピーすることができます。

```
Vector2 a(1,2);
Vector2 b(3,4);

Vector2 c=a;//値コピー発生(ついでに言うとコピーコンストラクタでコピー)
Vector2 d;
d=b;//値が代入され、コピー発生(代入によるコピー)
```

場合によっては、このような操作を防ぎたいことがあるとする。どういう場合に防ぎたくなるかというと、例えばシングルトンパターン(要はプログラム全体で一つしか存在を許されないクラスのこと)などは、「一つしか存在を許されない」ため、複製なんか

絶対に許さないよ

である。そういう時に使いたくなる。

さて、このコピー、代入禁止なのだが、作るのは非常に簡単なのである。まずはコピーコンストラクタを定義する。**コピーコンストラクタってのは初期化時に別のオブジェクトを右辺値に持ってくるときに呼び出される。**C++本だとゴチャゴチャ書いてるが、初学者はゴチャゴチャしてる所は読み飛ばせばいい。とにかく

```
Vector2 c=a;
```

こういう時にね。呼び出されているんだわー。っべーわー。

これを防ぎたい。簡単である。

private:　を利用するのである。privateで宣言したものは外側からアクセス出来ない。で、コピーコンストラクタの宣言はこう…

```
クラス名(const クラス名& 変数名){略}
```

である。いいんだ。コマケエコトを考えずにこいつをprivateにするんだ。それだけでコピー不可になるのだから。Vector2の霊ならば

```
private:
    Vector2(const Vector2& );
```

こう書くだけで
```
Vector2 c=a;
```

は失敗する。
ついでに言うと

```
return Vector2(rv.x+lv.x+rv.y+lv.y);
```

でコピーが発生するため、こいつも失敗するという泣きたくなる結果になるんだ(やっぱりVector2でコピー禁止はアカンな…)

で、

```
Vector2 d;
d=b;//値が代入され、コピー発生(代入によるコピー)
```

こいつはこいつで「値の代入」が発生するため、実質的なコピーが発生するので、こいつも防止する。
ここでオペレータオーバーロードの出番である。
何を使うのかというと
privateにoperator=を設定する。

これだけで代入できなくなる。

```
private:
    void operator=(const Vector2& );
```

これでオッケー。

もし、シングルトンパターンを使用するのであれば、**シングルトンクラス自身はコピー禁止、代入禁止**にしておこう。

さて、とりあえずコピー禁止、代入禁止が正常に動作することを確認したら、そいつらを一旦外した状態で
ベクタークラスを
ベクトル同士の足し算引き算、スカラー掛け算を実装して、結果のベクトルをコマンドラインを出力できるようにしてみよう。
掛け算演算子*では、内積を出力できるようにしてください。
ハット演算子^では、外積を出力できるようにして下さい。

ちなみにToString関数とか作っとくと便利かもね

```
std::string ToString(){
    std::ostringstream s;
    s << "(" << x << ","<< y<< ")";
    return s.str();
}
```

こんな感じでね。これについての細かい話は今は教えないので、興味アル人は各自Google

センセーに聞いておきましょう。結構役に立つですよ。
　もう一回言う。これについてはグーグルセンセーに聞きなさい。

よくわかんねー人のためにスクショを置いておきます。

```
                void operator+=(const Vector2& v){
                        x+=v.x;
                        y+=v.y;
                }
                std::string ToString(){
                        std::ostringstream s;
                        s << "(" << x << ","<< y<< ")";
                        return s.str();
                }
};
Vector2 operator+(const Vector2& rv,const Vector2& lv){
        return Vector2(rv.x+lv.x,rv.y+lv.y);
}

int operator*(const Vector2& rv,const Vector2& lv){
        return rv.x*lv.x+rv.y*lv.y;
}


int operator^(const Vector2& rv,const Vector2& lv){
        return rv.x*lv.y-rv.y*lv.x;
}

class A{
        const int nobita;
        A():nobita(1){}
};

using namespace std;
int main(){
        Vector2 a(2,3);
        Vector2 b(4,7);
        Vector2 c=a;
        Vector2 d;
        d=b;
        cout << (a+b).x << "," << (a+b).y << endl;
        cout << (a*b) << endl;
        cout << (a^b) << endl;
```

豆ちしきー

#pragma once

この授業の中ではたぶん使う人いないと思いますが、C/C++のプリプロセッサに

#pragma once

というのがあります。これは何かと言うと、**インクルードをした時に既にインクルードしているヘッダファイルを二回以上インクルードしない**(無視する)という機能をもつプリプロセッサです。

余談ですが#pragma once がなかった時代はどうしていたのかというと、ヘッダファイルの先頭で

#ifndef PLAYER_CLASS_HEADER_INCLUDED

#define PLAYER_CLASS_HEADER_INCLUDED

と書いて、末尾で

#endif

と書くことで、ここに挟まれたコードは二度目以降無視されます。これが何故こうなるのか気になる人は各自考えて下さい。ちょっと考えればわかると思いますので、想像力を働かせてみましょう。

ちなみに、二度インクルードすると都合が悪い理由は、以前にも言ったと思いますが、インクルード文と言うのは、インクルード書いた部分にそのままヘッダファイルがベコッと入り込みます。

このためクラス定義や構造体定義、変数宣言が二重になり、二重定義エラーが発生します。これを防ぐために#pragma once が有用なのです。

#pragma region～#pragma endregion

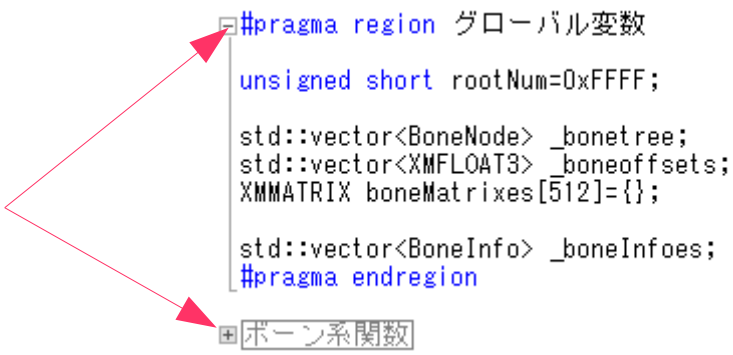
これはプログラム(実行ファイル)とは直接関係が無いのですが、たくさんコードを書いてプログラムが大きくなってきた際に役に立つ機能です。

ファイルを分けるのも面倒臭い場合に、一部をブロックにまとめてしまいたい時に使用します。

Visual Studio を使用しているならばこの#pragma region と#pragma endregion に挟まれた部分を纏めるための＋だかーだかが見えると思う。

ーと書いてあるなら、そこをクリックすればまとめて縮めることができるし、すでに縮められた部分は＋と表記されている。

以下にスクリーンショットを示す。



「グローバル変数」と表記している左横の-をクリックすれば、表記が+〔グローバル変数〕となり、表記がまとまるし、下の+〔ボーン系関数〕の左の+をクリックすると元の表記が復活します。

テンプレート

概説

テンプレート、これはいいものだ。ただしダークサイドに陥る危険性もはらんでいる。チョット使うくらいならいい。

だが、

http://ja.wikibooks.org/wiki/More_C%2B%2B_Idioms

に載っているテクニックや

「ModernC++Design」って本のテクニックを読むと間違いなくダークサイド行きである。もはや頭がおかしいレベル。

http://f3.tiera.ru/other/DVD-009/Alexandrescu_A._Modern_C++_Design(C)_Generic_Programming_and_Design_Patterns_Applied_(2001)(en)(271s).pdf

boostの中身を見ても、頭がオカシイことが分かるだろう。

テンプレートは便利すぎるために、使用法を誤るとKittyGuyになっちゃう。くれぐれも注意した上でこれからの話は聞いて欲しい。

テンプレートというのは型を特定しないで様々な関数やクラスを定義できるというスグレモノなのです。たとえば、2つの数のうち大きい方を返すような機能をほしいとする。このような機能は整数型だろうが浮動小数点型だろうが使ってみたい。

こういう時に使えるのだ。後で説明する関数テンプレートを使用すると

Max(1,7)は整数型の7を返し、Max(1.6f,9.0f)は浮動小数点型の9.0を返すことになる。

最後まで聞けば感じる人もいるけど、テンプレートは型を特定しないとかそんなチャチなものじゃ断じてねェ。もっと恐ろしい物の片鱗を味わうだろう。

関数テンプレート

先ほども言ったように関数テンプレートは型を特定せずにMax関数やらAdd関数やらを作れるものである。

で、ちなみに関数テンプレートの使い方は

```
template<class 仮の型名> 戻り値 関数名(パラメータ){
    ほにゃらら、ほにゃらら
}
template<typename 仮の型名> 戻り値 関数名(パラメータ){
    ほにゃらら、ほにゃらら
}
```

である。キーワードはtemplateとclassもしくはtypenameである。で、classとtypenameにはとりあえず違いはないと思っておいていいです。ですから好きな方を使いましょう。

たとえば先ほど例に出したMax関数を作るのならば

```
template<typename T> T Max(T a, T b){
    return a>b?a:b;
}
```

などという定義をすることができる。

で、仮の型名をここではTと置いてはいるが、何でもいい。で、数字や変数が入った時点で自動的にint型のMax関数やfloat型のMax関数が生成される。

つまり

```
cout<<Max(1,9)<<endl;
```

などと書いた瞬間に

```
int Max(int a, int b){
    return a>b?a:b;
}
```

が生成され、

```
cout << Max(1.0f,7.2f) << endl;
```

と書いた瞬間に

```
float Max(float a, float b){
    return a>b?a:b;
```

```
}
```

というコードが見えない所で生成されるわけです。ここまで読んだ人はもしかしたら「え？テンプレートって指定できる型名は1つだけなの？」と思うかもしれませんので複数設定してみましょう。

```
template<typename Ta,typename Tb> Tb Scaling(Ta value,Tb scale){
    return (Ta)value*scale;
}
```

なんて書いて

```
cout << Scaling(2,5) <<endl;
```

とでも書けば

```
int Scaling(int value,int scale){
    return (int)value*scale;
}
```

が生成されますし、

```
cout << Scaling(5,5.9f) <<endl;
```

とでも書けば

```
float Scaling(int value,float scale){
    return (float)value*scale;
}
```

というのが見えない部分で生成されます。ここに挙げた例だとあまりアリガタミがありませんが、ぼちぼち利用できそうな場面を見つけて活用…別にしなくてもいいですが、知識としては覚えておきましょう。C#でも「ジェネレータ」という名前で使用されますので、こういう「型がコンパイル時に決定される」系の話は頭の片隅に入れておきましょう。

クラステンプレート

関数テンプレートはまだわかりやすかったかもしれませんが、次は少しだけマニアックになってきます。なってくるのですが、実際はこいつが本番なんですねー。C++さんはホンマ初心者殺しやでえ…。

クラステンプレートってのは、要は**クラスのメンバの型をコンパイル時に決定する**というものです。

定義は

```
template<typename  仮の型名> class  クラス名{
        仮の型名 メンバ変数;
        仮の型名 メンバ関数();
        int a;
        void func();
```

```
};
```

と言った具合に定義します。ポイントは赤字で書いてある部分です。関数テンプレートの時と同様にtypenameまたはclassというキーワードで仮の型名を定義するところまでは同じで、クラステンプレートの場合、メンバに対して仮の型名を使えるというところが強力なのです。

例えば、前回使ったベクトルクラスはint型向けにつくりました。これをFloat型向けに作るとして、わざわざ

Vector2fなんてクラスを作りますか？作ってもいいですが、このクラステンプレートを使用すると簡単で。

```
template <typename T>class Vector2{
    public:
        T x;
        T y;
        Vector2():x(0),y(0){}
        Vector2(T inx,T iny):x(inx),y(iny){}
        void operator+=(const Vector2<T>& v){
            x+=v.x;
            y+=v.y;
        }
        (略)
```

などという感じに書き換えられます。これでVector2<T>というのはint型だけではなく、float型など様々な型に対して対応することができるようになります。

ということで本日の課題その1として、テンプレートによるベクトルクラスを完成させて下さい。つまり

配列オブジェクトをつくろー

実は今日の本番はこいつでしてね…。C++において、実行時に生成される配列をつくろうと思うと

```
char* c=new char[255];
```

などと書かねばならず、こいつにさよならするには

```
delete[] c;
```

などと書かねばならない。これはmalloc～freeでも同じですね。

どうせ関数の中だけで使用するような場合に動的配列は煩雑だし、解放忘れもある。どうにかして、通常の配列変数のように自動的に消えてくれないものだろうか…

なければ作るのがプログラマのお仕事です。

ということで、配列として作れば自動で消えるようなオブジェクトを作ってみましょう。もちろん、いろんな型に対応できるようにテンプレートも使用します。

ここで使用するテクニックは

- 配列の確保(new)、解放(delete[])
- 引数つきコンストラクタ
- デストラクタ
- クラステンプレート
- オペレータオーバーロード([]演算子)

です。今までやってきたことの集大成ですね。今回はこれを Array というクラスで作ってみましょう。

要求としては

- 配列のように使える
- スコープを抜けたら解放される
- 配列のサイズが分かる関数 Size()を作る
- 型を問わない

です。

```
template<typename T>class Array{
    private:
        T* _array;
        unsigned int _size;
    public:
        Array(unsigned int size):_size(size){
            _array=new T[size];//配列の確保
        }
        ~Array(){
            delete[] _array;//配列の解放
        }
        (これ以降は自分で考えて書いて下さい)
};
```

という定義があれば

```
Array<int> ints(100);
```

```
for(int i=0;i<100;++i){
    ints[i]=i+1;
}
```

というように使われ、スコープを抜けた後は、きっちり全て開放されるようにしてください。

これは int や float だけでなく Vector2 でも使えるものです。

```
Array<Vector2<int> > vectors(100);
for(int i=0;i<99;++i){
    vectors[i].x=i+1;
    vectors[i].y=i+1;
}
```

という風に使えるわけです。

コード的なヒントはここまでです。さぁ、頑張って作りましょう。

オペレータは[]演算子も使えますので、これを使って配列っぽく使えるようにして下さい。

戻り値& operator[](unsigned int idx)

などと定義すれば通常の配列の如く使えます。なお、戻り値に&が付いているのは、以前に勉強した「参照」を表しており、配列のように配列の中身を書き換えたいがためです。

とりあえずは、クライアント側で以下のように使えるようになってればオッケーです。

```
Array<int> ints(100);
for(int i=0;i<100;++i){
        ints[i]=i+1;
}
for(int i=0;i<100;++i){
        cout << ints[i] << endl;
}
cout << "size=" << ints.Size() << endl;

Array<Vector2<int> > vectors(100);
for(int i=0;i<100;++i){
        vectors[i].x=i+1;
        vectors[i].y=i+2;
}

for(int i=0;i<100;++i){
```

```
        cout << vectors[i].ToString() << endl;
    }

    cout << "size=" << vectors.Size() << endl;
```

このように使えるようになっていれば要求は満たします。これが今日の課題2だ。

頑張れ～。

スマートポインタ(初歩)

スマートポインタというのはその名の通り「スマートな(カシコイ)」ポインタの総称である。どういうことか。

例えばポインタの厄介な所はたとえばクラスEnemyがあったとして

```
Enemy* e=new Enemy();
```

とやると、当然sizeof(Enemy)ぶんの領域が確保されてしまい、**明示的にdeleteしない限り解放されることはありません。**

これは結構マズいことなんですよね。特に複数人でプログラミングしている場合はどうしても解放を忘れる人が出てくるし、そもそも解放のタイミングをいつにするのか、誰(どの所有者(クラス))が解放の責任を持つのか、予め決めておくか、自動で消えるようにして欲しいところです。

C#やってる人ならわかると思いますが、GC(ガベージコレクタ)があるだけで、随分とプログラムがラクになっているでしょう。

C++が嫌われC#が好かれるのはこういう所なんですよね？じゃあ何故未だにC++がゲームや組み込み、果てはAndroidにまで使用されているのかというと、ガベコレの解放タイミングがクライアント側で調整できない(ある程度の調整はできるが細かいのは無理)ため、意図しないタイミングで大量解放が行われてしまう。

コンピュータのメモリ解放はそれなりに時間的コストを食うのである。これをきっちり自分で把握したい要望が未だにゲーム業界や組み込みでは多いためC++が仕方なく使われているのである。まぁ、純粋にC++好きな奴も多い業界なんで…。

それはさておき、どちらにせよアホが居るチームで明示的なdeleteを期待するのは危険なことである。これをもうちょっとだけ安全に使う目的で作られるのがスマートポインタである。

スマートポインタはboostのが有名である。サーバーにboostからスマポライブラリ部分のソースコードだけ取ってきて置いていますので、興味がある人は見ておいて下さい。

スコープドポインタつくろー

スコープドポインタとは俺の造語…なのか？いやそんなことは無いと思うんだが……ほうぼうで「それ君の造語やん」って言われるのだ。

いやいや、だってboostの中にも"scoped_ptr.hpp"ってあるやん。まぁそれはさておき

配列オブジェクトまで書けていたらなんちゃないです。こいつは何かと言うと、通常はポ

インタで扱うべきものを通常の変数のように自動で解放されるものであるかのように扱えるようにするものです。

何を言っているのか分からねえと思うが俺にも何を言っているのか分かんねえじゃ仕方ないので、ぼちぼちと説明していきます。例えば

```
void function(){
	int a;//これはfunction()を抜けた時点でスコープから抜け、解放される
}
```

は問題なく変数aが解放されるのだが、

```
void function(){
	int* a=new int();
}
```

この場合、int型の4バイトがアプリ終了まで生き続ける…だれからも使われずにな!!更に言うと、これがメインループの中とかで呼び出されているのであれば非常にマズいことになる。おお…ヤバイヤバイ。

「そんな馬鹿な事やんねーよ」と思ってるあなた…甘いですよ。5人以上で開発してると、どこかにそういうコードが紛れ込むものだし、更に言うと1ヶ月まえの自分のコードは他人のコードである。

まぁ、とにかく少なくとも、関数内で確保したメモリは関数を抜ける時に解放したいもしくは所有者(所有クラス)が破棄される時に一緒に破棄されるようにしたい。

そんな場合に使用するのがスコープドポインタです。

ここで作るべきものは

- コピー、代入の禁止
- ->演算子オペレータオーバーロード
- *演算子オペレータオーバーロード
- デフォルト引数
- Reset()関数
- デストラクタでdelete
- ナマポを返すGet()関数

といった感じです。今回はboostのものよりは機能を少なくしておきます。

このスマポはスコープから外れると消滅する(関数内で定義されれば関数を抜けると共に消滅。クラスのメンバとして定義されていればクラスオブジェクト消滅とともに消滅する)哀れなスマートポインタです。でも結構使うと思います。

class ScopedPtrで宣言しておきましょうか

で、このスマートポインタは色々な型に対して使えるようにしなければ意味が無いのでテンプレートを使用します。

```
template <typename T> class ScopedPtr{
        T* _ptr;//本当のポインタを内包しておく
        :
}
```

実際に確保したメモリのアドレスを指し示すポインタをこのテンプレートクラスの中に内包しておきます。こいつが裏の主役となります。こいつに自動変数(いい子ちゃん)の仮面をかぶせて、善行をさせようと思います。

で、このスコープドポインタ、コピーや代入を行われてしまってはスコープドポインタが崩壊してしまうため、予めコピー、代入を禁止しておきます。

コピー、代入を禁止するにはコピーコンストラクタ及び=演算子を private にするのでしたね？

とりあえずコピーコンストラクタはこうでしたね？

```
ScopedPtr(const ScopedPtr& );
```

代入の方は前にやったので、自分でやっておきましょう。

次にコンストラクタにて真のポインタを代入→保持できるようにします。ですから

```
ScopedPtr(T* ptr=0):_ptr(ptr){}
```

とでも書いてやってコンストラクト時に真のポインタを渡せるようにしましょう。なお

```
ScopedPtr(T* ptr=0):_ptr(ptr){}
```

この部分ですが、C++ではデフォルト引数というものが使えるようになっており、この引数が入力されてない場合は=の右辺値が採用されることになっています。つまりこの場合は引数を入力しなければヌルポインタ(0)が設定されるようになっています。

そう、コンストラクト時に真のポインタが渡せない事も有り得ます。なんかしらの Create 関数を他所から呼び出して作らなければならない時なんかがそうでしょうかね。

ここで、遅延して真ポインタをセットできる関数 Reset 関数を作ります。

Reset 関数の仕様は

- 何もポインタが渡されなくて、かつ内包するポインタがヌルなら何もしない。
- 何もポインタが渡されなくて、内包するポインタが生きてるなら元のポインタを破棄して下さい
- なんかしらのポインタが渡されて、かつ内包するポインタがヌルなら新しいポイン

タを内包するポインタに代入して下さい。

- なんかしらのポインタが渡されて、かつ内包するポインタがヌルでないなら、元のポインタを破棄した上で、新しいポインタを内包するポインタに代入して下さい。

この仕様に合うようなReset関数を作って下さい。

さて、ポインタのようにこいつが挙動するのであれば->演算子でポインタ自身の関数をコールできなければなりません。

どうすればいいのか？簡単です。

->オペレータに元のポインタを返させればいい。それだけです。簡単すぎるでしょ？

では->オペレータオーバーロードを作って下さい。

次に、ポインタであれば*演算子で、ポインタのもつ「値」を返さねばなりません。これも簡単です。

自分の持っているポインタの値を返しさえすればいい。

というわけで*オペレータオーバーロードを作って下さい。

さて、ここまで書けたらスコープドポインタは出来上がっているはずですので

```
class Test{
    private:
        const char* name;
        int value;
    public:
        Test(const char* inname) : name(inname){
            cout << inname << "が生成されました。" <<endl;
            value=0;
        }
        ~Test(){
            cout << name << "が破棄されました。" << endl;
        }
        int Value() const{
            return value;
        }
        void Value(int v){
```

```
            value=v;
        }
};
```

こういうクラスを用意してやって、自分が作ったスコープドポインタを使用したサンプルコード

```
int main(){
    ScopedPtr<Test> a(new Test("a"));
    a->Value(6);
    cout << a->Value() << endl;
    cout << a.Get() << endl;
    a.Reset(new Test("b"));
    a->Value(2);
    cout << a->Value() << endl;
    cout << a.Get() << endl;
    return 0;
}
```

こんなのをつくってやって、

```
aが生成されました。
6
0x5b2e50
bが生成されました。
aが破棄されました。
2
0x5b3e78
bが破棄されました。
```

こういう出力が行われることを確認して下さい。**これが本日(11/14)の課題です。**頑張ってください。なお、アドレスを出力してる部分は各自のマシンで違うと思いますので、これと同じ値にはならないと思います。

簡単なリファレンスカウンタ(非スマートな)ポインタつくろー

スコープドポインタは哀れだったので、もう少しまともなものを作りましょう。ここで使用されるのは「参照カウンタ」という考え方です。

この「リファレンスカウンタ」が効果を発揮するのはあるポインタの複数の利用者がよってたかって使用するという状況で力を発揮します。
みんながよってたかって参照しますので、うっかり解放もできないわけです。でしょ？誰かの勝手な判断で解放しちゃったらマズいでしょ？かといって最後まで解放されないのもマズい…ですよねー？
じゃあどうしましょうか…ここで発想の転換です。所有者に削除させるのではなく…自分自身に削除させる…つまり自殺させるのです!!!
自殺…いけない響きだ、だが業界ではよく使われるフレーズです。ちょっとマジメな職場なら自律死だの自然死だの…まぁ結局縁起でもない言い方になります。
そして自殺コードはこうです。

```
delete this;
```

thisは自分自身を指す

今まで散々使ってきたかもしれませんが、thisってのはオブジェクト自身を表すポインタです。ポインタなのでdelete this;で殺せますし、thisのメンバを呼ぶ時はthis->Func();とか書きます。まあ滅多なことではthis->メソッド名()なんんてことはしませんがね。

とにかくここで覚えて欲しいのは

delete this;は自殺コードってことです。

さて、この自殺コードを示したのは何のためでしょう？そう、自分自身にオブジェクトを削除させるためです。では、削除のタイミングはいつにしましょうか？

「誰からも忘れられてしまうのなら、それは死んでいるのと同じことだ」

的な台詞をなんかの漫画で見たんですけどね。そういうことです。そのオブジェクトを持っていた誰もが「もう俺使わない」って言えば死んでいいのです。もしくは持ち主が全員死んだ時ですね。

ということで、何を死の基準にすればいいのでしょうか？そう、自分を覚えている人の数ですね？

自分を参照している人の数を覚えておこう

これが「参照カウンタ」です。つまり、この参照カウンタは「せっかくだから俺はこのポインタを使うぜ」と言われるたびにその数を増やしていき、「(°⊿°)ｲﾗﾈ」と言われるたびにその数を減らしていきます。
最後に減らされた時にこの数がゼロになったらもう解放してもいいというわけです。簡単ですよね？
ちょっとこの程度だったら自分で1から考えて作ってみて欲しいんですが…

```
class RefCountable{
    protected:
```

```
        unsigned int count;//参照カウンタ
    public:
        //参照の増加
        void AddRef(){
            //参照カウンタを増加させるコード
        }
        //参照の解放
        void ReleaseRef(){
            //参照カウンタを減少させて、その結果0になったら自殺するコード
        }
        virtual void Suicide()=0;//自殺
        ~RefCountable(){}
};
```

という感じの基底クラスを書いてあげて

```
class A : public RefCountable{
    public:
        //現在の参照カウンタを返す関数
        unsigned int RefCount()const{
            //現在の参照カウンタを返すコード
        }
        ~A(){
            //自らの死をコマンドプロンプトに出力
            cout << "I'm DEAD!!!" << endl;
        }
};
```

てな感じで派生させ、

```
A* a=new A();
a->AddRef();
a->AddRef();
a->AddRef();
a->AddRef();
cout << a->RefCount() << endl;
a->ReleaseRef();
a->ReleaseRef();
a->ReleaseRef();
a->ReleaseRef();
```

というコードをmainに書いてあげたら

```
4
I'm dead!!!
```

と出力されるようにコードを補完して下さい。どうやったらいいか分からない人は聞いて下さい。こんなのまだまだ入門にすらたどり着いてないので、こんなんでへこたれないでくださいね？

まあ、ここまで見ればDirectXのコードにやたらとObj->Release()ってコードが多かったのも何故だか分かってきてるんじゃないでしょうか？あれはそういうことなのです。
ここまでの話でとりあえず理解していただきたいことは、オブジェクトに自立して死んで欲しい時には裏に参照カウンタが隠れているということと、実際にboost等で実装されている「シェアードポインタ」の中身について大体のイメージがこれと同じということです。
このシェアードポインタを自作したいという人のために、もう少し後で説明します。今回作るのは簡易版リファレンスカウンタつきのポインタをテンプレ使って作ることを考えましょう。

テンプレ化

要は
```
template <typename T>
class RefCountablePtr<T> : public RefCountable{

};
```
ってなテンプレクラスを作りまして…スコープドポインタと同じような要領で作っていきます。*演算子とか->演算子のオペレータオーバーロードを行い、コンストラクタ時もしくはReset時に元のポインタを渡してあげます。
で、こいつのルールとしては呼び出し側にdeleteさせません。deleteはdelete this一択です。今回のやり方であればSuicideの実装によってdelete thisを行うことになります。が、実質デストラクタをprivateにするかどうかはT(型)側に任されるため、ここは強制はできないでしょう。…まぁ、簡易版ですから。
こんなクラスを作っておいて…
```
class B{
	public:
		void Out(){
			cout << "B dayo-"<<endl;
		}
		~B(){
			cout << "ore sinuno?" << endl;
		}
};
```
以下の様な感じでセットされる所はスコープドと同じですね。
```
RefCountablePtr<B> b(new B());
```
このコードはmain関数内にでも書いておいて下さい。
で、その後に
```
	b.AddRef();
	b->Out();
	b.AddRef();
	b.AddRef();
	(*b).Out();
	cout << b.Get() << endl;
	b.ReleaseRef();
	b.ReleaseRef();
	b.ReleaseRef();
```
というコードを書いた時に

4
I'm dead
B dayo-
B dayo-
0x8417f8
ore sinuno?
と出力されるような、RefCountablePtr<T>クラスを実装して下さい。とりあえずはスコープドで作った部分をコピペして、
あとはSuicide関数を実装すれば終わりです。
Suicide関数内では自分が持っている_ptrを削除します。(delete thisではございません。)
はい、ここまでが**今日の課題(11/21)**です。頑張ってください。

シェアードポインタの作り方

上の参照カウンタ。これでは実はスマートポインタとは呼べません。
本当に作りたかったのは「シェアードポインタ」というもので、概念的には参照カウンタで操作していくもので、発想は同じなんですけど、よりポインタっぽく使え、ユーザーにAddRefやReleaseなど呼ばせない…呼ぶ必要がないような作りにしなければなりません。

実際、これはしっかりしたものをつくろうと思うと吐くほどめんどくせえし、それなりに難しいです。この章はなんというか…FF5の「オメガ」とか「しんりゅう」みたいなもんだと思って下さい。わからんちんはここは読まないほうが良いでしょう。
わからんちんにとってこの章は禁書レベルでございます。ネクロノミコンでございます。
地獄を見たい人は一緒に作っていきましょう。
ちなみに僕は作ってます(マルチスレッド非対応ですが)。合計しても185行くらいでできますので、めんどくささはさほどではないのかもしれませんし、理屈とC++の仕様がわかってたら難しくもないかもしれません。ただ…そう思える人がどれくらいいるのか…拙者は心配にござりますので、まずは例外処理から読んどいて下さい。

ところで、volatile修飾子って知ってますか?C言語からある修飾子なんですが、これがまた殆ど資料が出てこないというクセモノなんですよ。C#にも受け継がれている修飾子なのですが、かなり特殊なものなのであまり話題に出ることはありません。

volatile修飾子とはヒトコトで言うと「特定の変数に対する最適化抑制」を行うものです。コンパイラが行う最適化ってどういうものか知っていますか?
コンパイラはReleaseモードとかにすると、無駄な処理(1回で済む処理を複数回やってるとか)を、なるべく計算回数が少なくなるようにしてくれるのが「コンパイラの最適化」です。

ただし、こいつにはチョイと難点があって…とくにマルチスレの時に難点があって、まぁ、コンパイラさんは最適化のために処理順を変えたりするわけです。うん、これはバカな僕らのサポートとしては非常にありがたい存在なんですが、おいそれと順序を変えてもらっては困る場合があります(特にマルチスレッド時)
こういうときのために、volatile修飾子を使うことによって、その予想外の最適化を抑制できるのです。

…まぁ、よっぽどプログラミングレベルが上がってこない限り意識する必要はないでしょう

し、使う機会も一生に数回あるかないかでしょう。でも、ときどきはvolatileのことを思い出してあげましょう。
まずは、参照カウンタ用のベースクラスを作る必要があります。
class CountBase
って名前ででも作りましょう。
で、中身にカウンタの…ポインタを持っておきます。うん、いきなりキタね。なんでポインタなのかって？そりゃあ君。複数のシェアドポインタから共有されるわけですからさ、「値」で持つわけにはいかんのよ。
と、するとやはり参照カウンタはintポインタ型で持つ必要がありますね。ではどのタイミングでnewすればいいのでしょうか？

- 最初の持ち主がnewする
- あとの持ち主はnewしない(既にカウンタ実体があるので)

で「あとの持ち主」っていうやつはどうやってポインタ共有するのかというと、基本的に「代入演算子」でポインタを共有するわけですね？
ということはキーになるのは

- =オペレータ
- コピーコンストラクタ

=オペレータは分かりますね？そもそも代入だし、当たり前です。では何故コピーコンストラクタなのかというと例えば

```
A a;
A b=a;//このときコピーコンストラクタが呼び出される。
```

ちなみにコピーコンストラクタは

```
A(const A& 参照元){
}
```

てな形で記述しますし、=オペレータは

```
void operator=(const A& 参照元){
}
```

です。

ということは、例えば参照カウンタを

```
int* _count;
```

と作っておいて、

```
CountBase(){
    _count=new int(0);
}
```

とでもしてやればいい。その上でコピーコンストラクタの時は

```
CountBase(const CountBase& c){
	_count=c._count;
	++(*_count);
}
```

とでもしてやればいい。

=オペレータも同様にする。ただし=オペレータの場合は注意が必要で既に値が割り当てられていた場合にはシェアドポインタの中身(ポインタ実体)が入れ替わりますので、前のは破棄したことにしなければなりません。ですから、

1.既にポインタが割り当てられているかチェック
	1.1割り当てられていたら元の参照カウンタを減らす
2.この時点でポインタが空きになってるので渡されたシェアポインタのポインタ本体を格納する
3.渡されたシェアポインタの参照カウンタを増やす

という手順になります。これでよし。なんちゃないでしょ？

これ以上の説明が面倒なのでソースコード書きます。

```
///参照カウンタに使用するカウントオブジェクト基底
///
///@note これをマルチスレッドにする場合は、インクリメントやデクリメントの部分を
///InterLocked等の関数で行う必要がある。('A`)マンドクセなのでちょっと今は考えない
///ただし、当たり前の話だが、カウント値も共有化しなければそれぞれが勝手に解放しやがる
///static変数を使うわけにもいかないし、さてイカがなものか…、動的にint型のメモリを
///確保し、Disposeのタイミングにて確保したintにも死んでもらうのが妥当だろう。
class CountBase{
	private:
		mutable volatile int* _usecount;//実際に使用されている数
		mutable volatile int _weakcount;//弱参照カウント(使用が０になったとたんに解放したらばダングリング参照の危険性あり)
	public:
		CountBase():_weakcount(0),_usecount(new int(0)){
		}
		CountBase(const CountBase& countbase):_usecount(countbase._usecount){
		}
		virtual void WeakAddRef(){
			++_weakcount;
		}
		///弱参照解放(弱参照カウントなので遠慮なく死んでいい)
		virtual void WeakRelease()const {
			if( --_weakcount==0){
				delete this;//自殺
			}
		}
		///派生クラス側のイベント
		virtual void Dispose()const=0;

		///参照カウント増加
		virtual void AddRef()const{
```

```
                ++*_usecount;
        }
        ///参照カウント減少
        virtual void ReleaseRef()const{
                if( --*_usecount<=0 ){
                        delete _usecount;
                        Dispose();//派生クラス側のイベント発生
                        WeakRelease();
                }
        }
};


///共有ポインタ
///他の変数に代入されるたびに参照カウンタが増え、
///このクラスが死ぬとともに、実体が解放される
//スマポはマルチスレッド時のことも考慮しなければ非常に危険である…
///が面倒なのと技術力が足りてないのでやってない
template<class T>
class SharedPtr :  protected CountBase
{
        private:
                T* _ptr;
                ///CountBaseから呼び出される純粋仮想関数の実装
                ///これが呼び出されたらこいつが持っている実体を解放する
                void Dispose()const{
                        if(_ptr){
                                delete _ptr;
                        }
                }
        public:
                SharedPtr(T* ptr=0):CountBase(),_ptr(ptr){
                        if(ptr){
                                AddRef();
                        }
                }
                virtual ~SharedPtr(){
                        if(_ptr){
                                ReleaseRef();
                        }
                        _ptr=0;
                }

                ///コピーコンストラクタ
                SharedPtr(const SharedPtr& sptr) : CountBase(sptr){
                        if( sptr.Get()==_ptr ){
                                return;
                        }
                        _ptr=sptr.Get();
                        AddRef();
                }


                ///代入
                ///@note 代入の際に自身が実体を持っている場合は
                ///それをいったん解放してあげなくてはならない。
                SharedPtr<T>& operator=(const SharedPtr<T> & sptr){
                        if( sptr.Get()==_ptr ){
```

```
                        return *this;
                }
                if(_ptr){
                        ReleaseRef();
                }
                _ptr=sptr._ptr;
                sptr.AddRef();
                return *this;
        }
        bool IsNull()const{
                return _ptr==0;
        }
        bool operator!()const{
                return IsNull();
        }

        ///完全リセット
        ///実際のポインタセット前に自分自身をリリースしとく
        void Reset(T* ptr){
                if(ptr==_ptr){
                        return;
                }
                if(_ptr!=0 ){
                        ReleaseRef();
                }
                _ptr=ptr;
                AddRef();
        }

        ///ポインタの先を指し示す->演算子のオーバーロード
        ///ここでなぜ手持ちのポインタをそのまま返しているのかというと
        ///->は後置演算子であるため、返されたポインタ＋右辺が指し示す
        ///ポインタを探しにいく。つまりこの場合、shared_ptr->のアドレスに
        ///クライアント側で指定している関数オフセットが加算されるために
        ///メンバ関数が正しく呼び出され、メンバ変数を正しく参照できる
        T* operator->() const{
                return _ptr;
        }

        ///参照を返す
        T& operator*() const{
                return *_ptr;
        }


        ///生ポインタを返す
        T* Get()const{
                return _ptr;
        }
        ///持っているポインタを交換する
        void swap(SharedPtr<T>& ptr){
                T* tmp=ptr._ptr;
                ptr._ptr=_ptr;
                _ptr=tmp;
        }
};
```

例外処理

Javaだか C#だかやってたら、try～catchとか言うのを目にしたことだろうと思いますが、それはC++でも存在します。
実はコンシューマゲームでは殆ど使われていないのが現状です。これには歴史的背景とかもあるんですけどね……よく言われる理由としては「遅い」ってのがありますし、あとは昔のコンパイラのtry～catchの実装がショボかったっていう悲しい過去があります。

モウヒトツ言えるのはアーケードゲームの場合「予期せぬエラー」が出た時点で「もう進行不可能」としWatchDogってな仕組みが走って強制的に再起動される仕組み(これは今でもそう)になっているため、「例外処理をしてもムダ」ってな思想になってたんですね。

コンシューマでも程度の差こそあれ、似たようなものですし、MSのDirectXのサンプルコードでも殆どtry～catchが使われていないのも原因でしょう。

とにかく色々と理由があって殆ど例外処理が使われなくなってますが、これからの開発ではまた復活しそうになってきてますし、ソーシャルだのITだのを狙うなら知っておかねばらないのが、この例外処理であるというのも事実です。

前置きが長くなりましたが、早速例外処理について学んでいきましょうか…。

try～catch構文

C++には例外(エラー)を投げる仕組みとキャッチーするためのしくみがあります。というわけで、わざとエラーを起こすコードを書いて、それをキャッチしましょう。
さてさて、いろんなエラーが存在するわけなんですが、例外ってのは基本的には自分で(実装側で)発生させなければ、誰も起こしてくれません。

どういう事かというと、100/0.fなんていうゼロ除算エラーとか出してくれませんし、バッファオーバーランも基本的には例外を発生してくれません。
なので、基本的にはこの後で説明するthrowキーワードを使用して「エラーが起きたぞー」って通知する必要が有ります。

ただ、いきなりthrowを自分で書いておきながら、それをキャッチしてしまうとなんだか「自作自演」な感じがするので、ひとまずはSTLに標準で搭載されているエラーを出してみましょう。

はい、まずはtry～catchの書き方から説明します。書き方としては

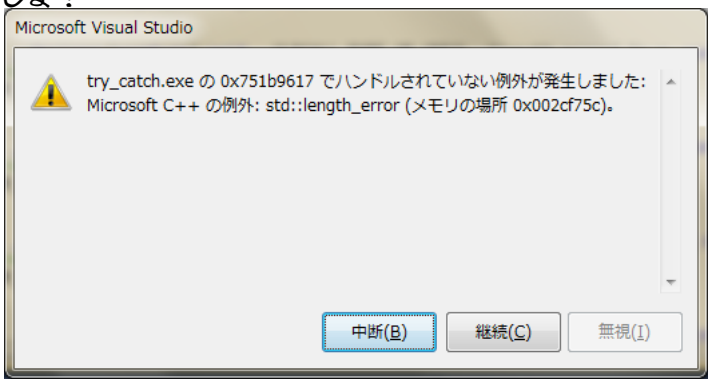
```
try{
	チェックしたい文;
}catch(例外クラス){
	エラー報告等;
}
```

基本的にはチェックしたい処理を全部try{と}でくくってしまいます。そうすれば、なんかしらエラーが起きた際に、即時にcatchの後の{カッコ}の中の処理を実行します。
よくわからない人もいるかもしれませんので、わざと発生させてみましょう。STL自身が発行する例外を捕まえてみます。
とりあえずvectorを使いますので、#include<vector>しといてください。で、標準のエラーをキャッチするには#include<stdexcept>もしておく必要があります。
最初に紹介するのは…そうですね。bad_allocを紹介します。bad_appleじゃないんで、東方厨さんは間違えないようにお願いします。

ひとまず、これらをインクルードした後でいつものようにメイン関数を作って

```
vector<int> v(-1);
```

と書いて下さい。明らかに間違いなんですが、エラー起こすのが目的なんで気にしないでください。
書けましたか？
実行してみてください。
例外…発生したでしょ？



こんなの。
で、こんなの出すのは構わんのですが、皆さんがなんかしらのアプリケーションを作ったとして、こんなんお客様に見せても困るわけです。
なので予めエラーが発生する前提で罠を仕掛けておくわけです。
このしかける罠ってのがtryなんですよ。

というわけで、さっき書いたコードをtryカッコでくくってくだしあ。
tryでくくっただけではコンパイルが通りません。こいつはcatch構文とペアで使用します。
で、すまん、さっきbad_allocの紹介をすると言っていたがあれは嘘だ(結果的に)。よく見てもらえばわかるが、length_errorが発生している。
では、こういう場合のcatch構文の書き方を教えておこう

```
catch(const length_error& err){
    //エラー処理を書く
}
```

という書き方になります。これでtryでクラッシュするのではなく、catchに処理がジャンプします。
ここでエラーに対して適切な処置を施します。
まぁ今はお客様いないんだから

cout << "キエエエエエアアアアアアアエラッタアアアアアアア!!!" << endl;

でもいいんですが、何でエラッタのか教えてあげるのが親切ですね。

cout << "長さ的におかしくない？" << endl;

とでも書いておいてあげましょう。
で、このerrって変数は何なんでしょう？ここにはエラッタ時の情報が入っています。ですから
err.what();
という関数を呼ぶと、エラッタ時の詳細状況が返ってきます。ということで

cout << err.what() << endl;

と書いてやると

"vector<T> too long"

などという出力が得られます。

では次に-1ではなく、400000000とでもしてみてください。そうすると今度は別の例外が出ます。今度こそbad_allocですね。これマシンによっては確保できちゃうんで発生しないかもしれませんが、そういうときはこの数値を増やしておいて下さい。増やしすぎるとlength_errorになっちゃうので注意です。
今回のエラーはlength_errorでは捕まえきれませんので、bad_allocを仕掛けてやる必要があります。
ですから

```
try{
    vector<double> v(400000000);
}catch(const length_error& err){
    //長さエラーの対処
}catch(const bad_alloc& err){
    //メモリ確保エラーの対処
}
```

となります。

このようにクラッシュするだの不親切なダイアログボックスを出すのではなく、クライアントに**親切なメッセージを表示**し、かつ**アプリケーションを止めない**ってのがシステム開発では重要な要件となります。
バグって炊飯器止まったらイヤでしょ？お金関係とかだったら取り返しがつかないでしょ？というわけで、例外処理ってのはIT系やるなら知っとかないといけない仕様です。

はい、とりあえずその他にも色々とありますので体験しておきましょう。

vector<int> vec(1);

vec.at(1)=9;

とか書いて下さい。実行すると当然のようにout_of_range例外が発生しますので捕まえてごらんなさい。

ちなみに

vec.at(インデックス)

ってのは、意味的には

vec[インデックス]

と同じ意味なんですけど、後者の場合だと、例外を発生してくれないんですよ。例外を発生せずに何をするのかというと、バッファオーバーランして、メモリ破壊しまくります。ですから、システム開発系の時はvec[インデックス]よりvec.at(インデックス)のほうがいいでしょうね。
結果は

"invalid vector<T> subscript"

なんて出力されます。

次に

```
class Base
{
public:
    virtual ~Base(){}
};

class Derived : public Base
{
};

void func(){
    Base base;
    Derived& der = dynamic_cast<Derived&>( base );
}
```

とか書いて、func()関数を呼び出してみてください。
見事にbad_cast例外が発生します。これもやってみてください。
ここのdynamic_castについては、もう少し後で説明する「いろんなキャスト」で説明しますので、ここでは(' _ >`)ﾌｰﾝくらいに思っといて下さい。

Bad dynamic_cast!!!

とかって言われると思います。とりあえずはここまで全て体験してみてください。ちなみにC++の仕様にはfinallyってのはないっぽいです。残念。
ちなみに、これらのエラー全てに対応するのが

```
catch(...){
}
```

です。やってみましょう。
全ての例外はエラッたらこのcatchで捕まえられます。捕まえられますが、有用な情報がないので、非常にこまりますが、何がくるのか予想つかない場合の最後の手段として覚えておきましょう。
これと同じ働きをするのが…

```
catch(const exception& err){
}
```

です。
どういうことなのかというと、全ての標準エラーはstd::exceptionから継承されているからです。これで全部まかなえるっちゃまかなえますが、それぞれのエラーによって後処理が違うことが殆どですので、予想できるものは出来る限り分けておくことをお勧めします。

throw

はい、で最初にも言いましたが、例外処理ってのは基本的に自分で投げるものです。標準C++は投げてくれません。STLが投げてくれるから、投げてくれそうな気がするだけです。

例外を投げるためのキーワードはthrowです。

というわけで投げてあげましょう…そうですねゼロ除算を防ぐものを作ってみましょうか。

例えばこういう関数

```
float divide(float lval,float rval){
    return lval/rval;
}
```

を作っておいて、

```
cout<<divide(5.f,0.f)<<endl;
```

なんてやると、エラーは出ないですが、変な出力INFなんていうのになると思います。こういう時に例外を発生させたい場合は

```
    if(rval==0.f){
        throw;
    }
```

なんて書けば例外をここで発生させることができます。

但しこいつは「キャッチできない例外」となります。キャッチしたければなんかしら情報を渡してやります。

```
throw "zero divided";
```

これで今度はcatchの方で、文字列を受け取るようなcatchを書いてやればいい。文字列を受け取るcatchは至って簡単。

```
catch(const char* err){
}
```

でいいです。とりあえずゼロ除算したら例外発生させて、その例外をキャッチさせてみてください。

ただしこのままではexceptionでまとめて捕まえるなんていう芸当はできませんのでそれがしたければ継承して投げます。

exceptionから継承して…

```
class ore_error : public exception{
public:
    ore_error():exception("zero divided", 1){}
};
```

でこいつのオブジェクトをなげたら、さっきと同じような捕まえ方ができます。やってみてください。

RTTI(実行時型情報)

実行時型情報とはなにか…？とりあえずポリモーフィズムは覚えてますね？あ、覚えてない？そいつは困った…。同じ親から継承されている別々のクラスを同じように扱えるというアレです。

普通、**型というのはコンパイル時に決定してしまう**ものですが、ポリモーフィズムを使用している場合は、実行時にならないと型がわからないことがあります。

例えば Animal クラスを作っておいて

```
class Cat:public Animal
```

だとか

```
class Dog:public Animal
```

だとかを作ってしまいます。そうすると

```
Animal* animal1=new Cat();
Animal* animal2=new Dog();
```

と言った具合に、animal は Cat かもしれないし Dog かもしれない。挙動を見ないとわからないわけです。

この Cat か Dog かどっちなのかってのが「**実行時型情報**」なわけです。

ごちゃごちゃ言うより、見てもらったほうが早いですね。

とりあえず、

```
class Animal{
    public:
        virtual void Func(){
        }
};
class Cat : public Animal{
    public:
        void Func(){
        }
};
class Dog : public Animal{
    public:
        void Func(){
        }
};
```

とでも書いて下さい。

例えば、

Animal* ani;
(中略)

ってあったとして、確かに中身がある。…なんて場合に、最終的な中身の型を知りたい時の強うい味方がC++にはあります。

そいつの名前はtypeidまぁ、それなりに役立つやつさ…まぁ、使わないことも多いんですがね。

typeidについて

C++には実行時型情報を知るためのキーワードとしてtypeidなんていう演算子が有ります。演算子の一つなのでsizeofと同じです。
こいつはsizeofがsize_t型のなにかを返すのと同様に、typeid演算子はtype_info型のなにかを返します。まぁ、とにかく型を表すものです。

専門的なことはともかく

Animal* ani;
と宣言しておいて
Cat c;
Dog d;
と、Animalから継承した型で変数宣言します。
ani=&c;

だとか

ani=&d;

だとかで、中身を入れ替えることができます。
まぁ、とにかく、中身を入れ替えながらtypeid演算子を使うわけですが、使い方は

typeid(変数名);

とか

typeid(型名);

で、それぞれの型情報を取得することができます。

ちなみに

typeid(ani).name()

とやると、型名を表す文字列が返りますので、出力してみてください。

で、中身をCatだのDogだの変えながら出力してみてください。

あれれ？全部"class Animal*"ですよ？

ええ、そのとおりです。Animalポインタ型ですから。ですからこうしてみてください。

```
typeid(*ani).name();
```

どうでしょうか？

"class Cat"

だの

"class Dog"

だの出力されましたか？
そう、外側から見ればみんな同じに見える

*ani

の中身の型を見れるのです。これは楽しい。更にこいつを使えば中身がCatのときとDogの時で場合分けができます（正直これで場合分けする時点でなんか間違ってる気がするが）。

```
if(typeid(*ani)==typeid(Cat)){
}
```

などで型の比較をすることができます。これはtype_info型が==演算子と!=演算子をオーバーロードしてくれているからです。ありがたいですね。
しかしここでちょっと注意。

Animalのvirtual関数を消してみてください。

…

結果が変わりませんでしたか？そう、こいつはポリモーフィズム状態になっていないと使えないものなのです。どういう事かというと、このtypeid演算子はvtblを見に行ってこの情報を取得するためです。

まぁ、とにかくこういうものだということは今日覚えて帰って下さい。
とりあえず、今日の課題は、型で場合分けして、挙動を変えるということをやってみてください。

いろんなキャスト

C++には色々なキャストが存在します。殆どの場合はC言語の"(型名)変数"ってなキャストで事足りるんですが、プログラムを明確化していく上で「このキャストはどういう意図を持ったキャストなのか」ってのを他のプログラマに明示する意味が強いですね。

static_cast

一番良く使う…かな？int⇔floatの変換などに使います。C言語の通常キャストに最も近いといえるキャストです。

static_cast<変換したい型名>(変換したい変数)

といったフォーマットで使用します。static_castは結構厳しいキャストです。名前から推測できると思いますが「**静的な型**」に適用されるキャストです。ですから動的な型、つまり以前に紹介したRTTI(実行時型情報)がからむキャストには不適切です(キャスト自体ができないわけではない)。

静的な型キャストってのは例えば

```
float a=3.141592;
int b=static_cast<int>(a);
```

のように、コンパイル時に型が分かっているものに適用されます。で、一応C++の仕様を読むとstatic_castは

- あらゆる型からvoidへの変換
- 基本型から派生型の参照への変換
- 一部の標準変換の逆変換

をサポートするとあります。

「あらゆる型からvoidへの変換」…なんだこれ。

```
static_cast<void>(5);
```

とかが許可されるとある。確かにこれ、一見怒られそうだけどコンパイルは通ります。試してみてください。

ただしよく考えて下さい。void型の変数など存在しないのです。

つまりこのキャスト自体には意味が無いわけです。

ではこの規定に何の意味があるのかというと、関数テンプレートの特殊化の際や、明示的に関数の戻り値を使わないことを明示するために使う…みたいですが、たぶんそっち理解するほうが難しいので、とりあえずそういう規定があるということだけ頭の片隅あたりに置いておいて下さい。

では次に基本型から派生型へのキャストですが、例えば

```
struct Base{
};
struct Derived : public Base{
};
```

と言ったクラス構成であるとします。そうした場合、

```
Derived derived;
Base& base=derived;
```

は当然ポリモーフィズムルールのためこの変換は可能です。ちなみに暗黙の型変換というやつが見えないキャストをかけてくれるためわざわざキャストする必要はありません。ただし、これが逆だとエラーを起こします。

```
Base base;
Derived& derived=base;
```

当然ですね。base から派生した derived の方が基本的には変数も関数も多いわけで、それを派生側に変換したとすればメモリ的にヤバイ事になります。しかし…この static_cast の規定によれば**「基本型から派生型の参照への変換」は可能**であるということです。

つまり…

```
Base base;
Derived& derived=static_cast<Derived&>(base);
```

これが可能となってしまうわけです。危うしメモリ!!…とまぁ、こういった融通がきいちゃう部分が**「静的でない型には不適切」な理由**なんですが…とはいえ、なんでもかんでも通すってわけではなく、例えば

```
class A{};
```

っていう全然違う、関連性のない1型があるとして、

A& a=static_cast<A&>(base);

ってのは通りません。これが可能なのはreinterpret_castだけです。

const_cast

constはずしに使う。constはずしとかわからんよね…どっからconst参照で返されたものを、なんかしらの理由でconstはずしをしたい時に使用します。

例えば

```
const int a=13;
int& b=a;//バッカおめえ何constはずしやってんの？
```

これはダメです。何のためにconstにしてると思ってるんでしょうか…バカですねー。ということでコンパイラが激おこです。

これにはstatic_castも効きません。

```
const int a=13;
int& b=static_cast<int>(a);//激おこ(●`ε´●)
```

ところが…ですよ。ここにconst_castというキャストをしてやることにより、このエラーを回避できます(ホントは回避しちゃいかんのだけど)

しかし、あくまでもエラー回避のためだけにある機能だと思っておいて下さい。実は他所の関数とか使ってる時にconst参照返しやがって、使えない時が多々ある。そういう時に緊急回避的に使うキャストです。

ですから、上の例で言うと

```
b++;
cout << b<<endl;
cout << a << endl;
```

なんて書くと、どういう結果になるのかというと

14

13

と出力されます。

わかりました？参照なのにコピー状態になってるってわけです。constはあくまでもconstなんですね。

dynamic_cast

ダウンキャスト…前に説明したRTTIと関係してる。ダウンキャスト不可のときにbad_cast例外を発生させてくれる。これはstatic_castの時に

```
Base base;
Derived& derived=static_cast<Derived&>(base);
```

をやっても特に何もおきませんでしたが、ここをdynamic_castにするとRTTIを見に行って、キャストが適切かどうか判断してくれます。判断の結果としてアウトなら、**bad_alloc例外を発生してくれる**カシコイやつなのです。

ですから、ポリモーフィズムによってDerivedからBaseの参照に変更できますよね？その場合、たまーに一度Baseとして扱ったものをDerivedに戻したい場合があります。どういった場合かというと、例えば

```
std::vector<Base*> _bases;
_base.push_back(new Derived());
```

としておいて、例えば

```
class Base{
	public:
		virtual void Update(){}
};
class Derived : public Base{
	private:
		int _koyuchi;
	public:
		Derived(){_koyuchi=6;}
		void Update(){}
		void DerivedFunc(){
			cout << "派生側固有関数" << endl;
			cout << "派生側固有値=" << _koyuchi << endl;
		}
}
```

このへんの話は本当はものごっつ大変な話なのですが、とりあえず使えるやつだってことは分かっておいて下さい。

```
_base[0]->DerivedFunc();
```

というふうに、派生側の関数を呼びたい時があります。当然、Base 型には DerivedFunc はありませんから？

```
(Derived*)(_base[0])->DerivedFunc();
```

とでも書かなければイケナイわけです。こういう風に基底クラスから、派生クラスへのキャストのことを

ダウンキャスト

と言います。この用語は覚えておいて下さい。少なくとも C++ を使う職業につく可能性がある人は覚えておきましょう。いつ使うかわかりませんよ？

僕もそれまで使ったことがない Delphi とかいう言語を一度だけ開発することを強いられて、それ以降使ったことがないんですが…会社に入ったら強いられることもあるのです。ま、頭の片隅に置いておきましょう。ゲーム外車行く人はガチで知っておきましょう。

で、上の C キャストだと static_cast とほぼ同じなので、**不適切なダウンキャスト**も通しちゃうんですけど、dynamic_cast はそれも検査してくれるってわけ(ただし実行時に限る)。

もちろんさっきの例の

```
dynamic_cast<Derived*>(_base[0])->DerivedFunc();
```

は通ります。そして

```
_bases.push_back(new Base());
dynamic_cast<Derived*>(_base[1])->DerivedFunc();
```

も通ります。通るんですけど、RTTI から判断した結果……dynamic_cast は NULL を返します。以前に bad_cast 例外が発生するとは言いましたが、相手がポインタの場合は NULL 返しや済むんですから、いちいち例外なんて必要ないんです。

これが参照になると…

```
Base base;
Derived& derived=static_cast<Derived&>(base);
```

これは通りません…当然ではあるんですけどね。ポインタの場合はNULLが返せますが参照はNULL返せませんのでbad_cast例外を発生せざるを得ないわけです。

dynamic_cast<T>(value);

のTはポインタ型か参照型でなければいけません。値型はダメです。

reinterpret_cast

なんでもおっけー。うぇるかむとぅカオス。**使わないに越したことはない**お(^ω^)。これもstatic_castの時に例を上げましたが、結構変換してくれるstatic_castであっても

```
A& a=static_cast<A&>(base);
```

Baseと全然関係のないAってなクラスにキャストしようとすると、さすがのstatic_castさんも黙っておれなかったわけですが、

このreinterpret_castは、それを可能にします。**C++において基本的には使わない方がいい機能の筆頭に上がるキャスト**です。とはいえ、そういうのが必要な場合もあるっちゃあるので、頭の片隅にでも置いておいて下さい。

こいつを忘れてもいい世の中になる事を切に願っております。

まぁ、あえてありうるとすれば

- ポインタから整数型への変換
- 整数型又は列挙型からポインタ型への変換
- オブジェクト、関数、またはメンバを指すポインタ同士の変換

でございます。
「ポインタから整数型への変換なんかしねーよ!!」はい、そのとおりでございますが、実はアドレス値を「ハンドル(識別するもの)」として扱いたい時がありますので、そういった時に整数型に変換してunsigned int型の変数に代入しちゃうことはあります。

整数型または列挙型からポインタ型への変換。

…やっちゃダメだろとは思いますが…まぁ最初の例でハンドルにした後で元に戻したい時もある…てな感じでしょうか。

おまけ

最後にキャストに関して、ちょっとお話。

例えば

```
float a=(float)16;
```

これは普通にキャストですよね？実は

```
float a=float(16);
```

とも書けます。これは厳密に言うとキャストではなく、初期値16のfloatオブジェクトをaに代入しているという式になっているわけです。まぁ、キャストみたいなもんなんですし、こういう書き方をする人もあまりいないのですが、これだけを見てキャストの書式が**型名(値)**って思っちゃうと思わぬ所でハマりますので注意しておきましょう。

メンバ関数ポインタ

関数ポインタおさらい

ここでちょっと確認なんですが、C言語の**「関数ポインタ」**って知ってますか？知りませんか？そうですか…**関数ポインタ**ってのは、C#でデリゲート使ったことあるかな？あれにちょっと近い。

実は関数自体もメモリのどっかしらに配置されているわけで、そいつはアドレス管理されています。アドレス管理されているってことは、変数に代入できるわけでしょ？

変数に代入できるってことは**代入しなおしができる**ってことです。

代入しなおしってのは例えば

```
int a=17;
a=20;//17が入っていたaの中身を20で上書き
cout << a << endl;//当たり前ですが"20"が出力されます。
```

今のキミタチにとっては当たり前ですね？これ、関数でもできちゃうわけです。

たとえば

```
void Punc(){
    cout << "オラオラオラオラオラオラオラ" << endl;
}
void Kick(){
    cout << "無駄無駄無駄無駄無駄無駄無駄無駄無駄無駄無駄無駄" << endl;
}
```

関数ポインタ func=Punch;

func();

func=Kick;

func();

ってやってそれぞれの挙動を示すとすれば、状態遷移等に使えそうでしょ？

…なんかに似てると思いませんか？そう、ポリモーフィズムに似てますな。で、上でもったいぶって「関数ポインタ型」なんて書きましたが、具体的に書くと

戻り値型 (*変数名)(パラメータ)

これが関数ポインタの宣言です。こう書くと変数宣言っぽくないんですが、上の例で言うと

void (*func)()=Punch;

こういう記述になりますし、

func=Kick;

はそのままでいいです。宣言がヤヤコシイだけです。で、呼び出しはというと

func();

とやればそのまま呼び出せます。通常の関数呼び出しとおなじ感覚で呼び出せばいいのです。

ところで宣言がどうしても「変数の宣言」っぽくないので、プログラマはよく typedef を用いてあらかじめ「関数ポインタ"型"」を定義したりします。

typedef 戻り値(*型名として扱う名前)(パラメータ);

と書いておけばあとは変数と同様に関数ポインタの宣言ができます。

typedef void(*Function)();

と宣言しておけば先程の例では

Function func=Punch;

func();

func=Kick;

func();

と言った感じで使えるようになります。普通の変数みたいでしょ？

今回は引数も戻り値もありませんでしたから引数と戻り値入れてみましょう。例えば攻撃回数を引数として受け取ってヒット数を返す関数とすると…。

```
int Punch(int attackCount){
    cout << "パンチ回数=" << attackCount << endl;
    return attackCount*2/3;
}
```

```
int Kick(int attackCount){
    cout << "キック回数=" << attackCount << endl;
    return attackCount/2
}
```

という関数が考えられます。例えば

```
typedef int(*Function)(int);
```

と定義しておけば、

```
Function func=Punch;
cout << "ヒット回数=" <<func(6) << endl;
func=Kick;
cout << "ヒット回数=" <<func(9) << endl;
```

とでも書けば引数つきの関数の結果の戻り値を使うこともできます。

また、昔のゲームの現場では「タスクシステム」というのがよく使われており、状態遷移その他のために面白い手法が使われておりました。例えば以下のように構造体を定義しておきます。

```
class Task{
    void (*func)(Task*);
    char work[64];
};
```

てな関数を作っておきます。なお、workってのは連続メモリ64バイトを使うって意味で、中で何にキャストするかは各関数に任されます。

で、自分自身に「実行すべき関数」を持つことによって、関数それ自信が処理のバケツリレーを行うわけです。例えば

```
void Normal(Task* task){
    if(ダメージ受けた){
        task->func=Damage;
    }
}
void Damage(Task* task){
    if(体力＼(^o^)／オワタ){
        task->func=Dead;
    }
}
```

```
void Dead(Task* task){
	if(60フレーム待ち){
		task->func=NULL;
	}
}
```

で、ゲームループが例えば

```
task=new Task();
while(task->func!=NULL){
	task->func();
}
```

とでもしてやるわけですよ。そうすれば、各関数の中に自律して次の状態のことを記述できるので、プログラムが見やすくなります。

…これが関数ポインタなんですが、これはあくまでも事前説明に過ぎない。本番は「メンバ関数ポインタ」である。

メンバ関数のポインタ

上の「関数ポインタ」を理解した人でも、メンバ関数のポインタになるとちょっと注意が必要なのだ。基本的な利用価値は変わらないのだが、宣言も実行も違う。これに注意しないと「ああ、C++のクラスでは関数ポインタ使えへんのや」と思うことになります。

今はネット調べりゃすぐわかるんですが、数年前まではそうそうこの情報もなかったので誤認識が結構あったみたいです。ですから職場によっては未だに「C++でメンバ関数ポインタは使えない」とか思われてる事があるので、注意しましょう。

さて、メンバ関数ポインタなんですが、宣言がそもそも違います。こうです。一部違うんですね。

通常関数ポインタは

```
void(*func)();
```

という宣言でしたが、メンバ関数ポインタはわざわざ「誰のメンバ？」ってのを明記しないといけません。

戻り値（クラス名::*関数ポインタ)(パラメータ)

面倒ですね。

ですから例えば

```
class Player
```

という関数であれば

```
class Player{
	void(Player::*func)();
```

};

と宣言する必要があります。さらにヤヤコシイのは、関数の代入なのですが、通常の関数ポインタであれば

関数ポインタ型変数=関数名;

で代入できましたが、メンバ関数ポインタの場合は

関数ポインタ型変数=&クラス名::メンバ関数名;

となります。ホントに面倒です。先程の例で言うと

func=&Player::Normal;

と指定します。これでメンバ関数 Normal がメンバ変数である func に代入されます。そしてそして最もヤヤコシイのが呼び出し…。

通常の関数ポインタであれば、

関数ポインタ名(パラメータ);

たとえば

func();

ってな感じで呼び出せましたが呼び出しがなぁ…まぁいいんですが、呼び出すとすればこうです。

(this->*func)();

と呼び出します。

もちろん、パラメータを与えたければ

(this->*func)(パラメータ);

になります。

ここで注意すべきなのは->*演算子です。メンバ関数ポインタ以外でほぼお目にかかる機械のない演算子です。ちなみに持ち主がポインタ型でないばあい…この場合ムリヤリするなら

((*this).*func)(パラメータ);

と言った具合になります。.*演算子です。ヤヤコシイです。ちなみに、->*演算子や.*演算子もオペレータオーバーロードができます。

…まぁ、これオペレータオーバーロードしてもウレシイ事はないと思いますので、「できる」ってのを頭の片隅にでも置いていただければ結構です。

ちなみにワタクシはこのメンバ関数ポインタをどこで使っているのかというと、プレイヤーだの敵だのの状態遷移に使ってます。

通常→やられ→死にモーション中→死→復活モーション→通常

てな感じに移行できるように、各状態を関数化して、その関数内で条件チェックして次の状態に移行させてます。

また、最後になりましたが、関数ポインタもメンバ関数ポインタも、変数ではあるので配列に入れることも可能です。

例えば

```
Function funcs[]={Punch,Kick,Jump};
```

といった具合ですね。ゲームによっては役に立つと思いますので(例えば格ゲーで攻撃方法がランダムとかね)、役立てていくのもいいでしょう。

ただこれを知らなくてもプログラム自体は組めるしゲームも作れるので、わかんねー場合は無理して使うものでもないでしょう。

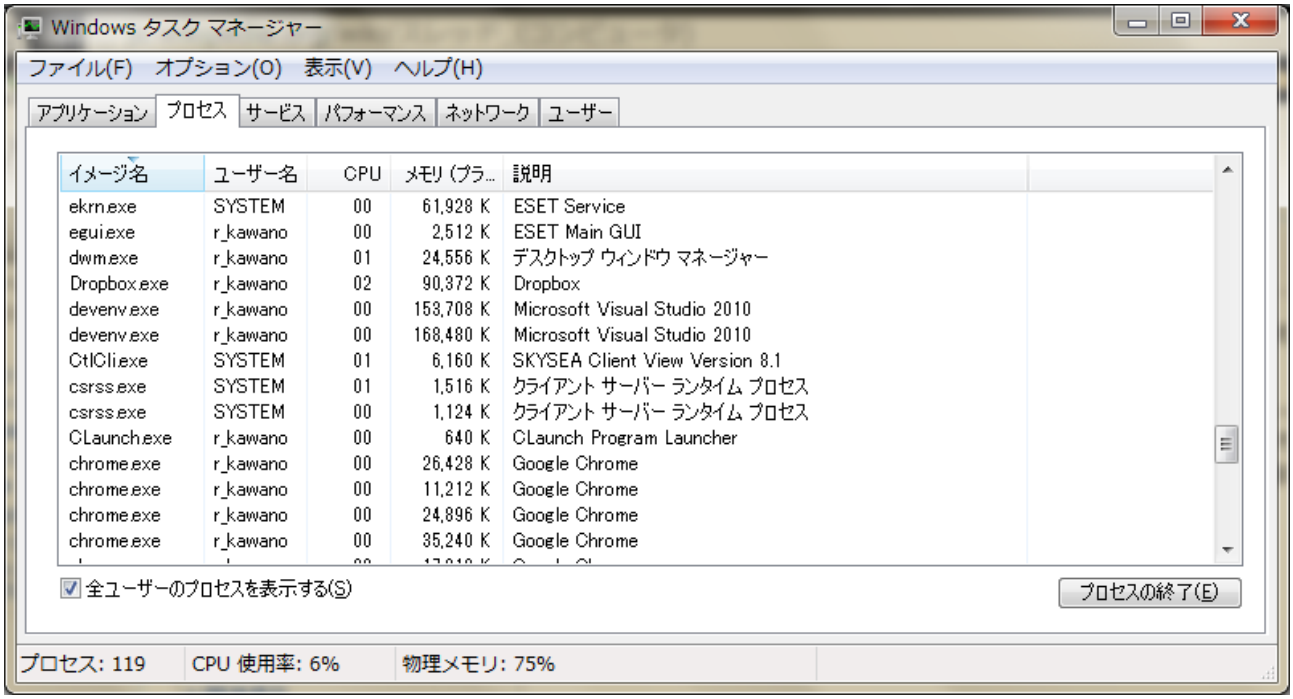
スレッドプログラミング

お待ちかねのスレッドプログラミングです。キマシタワー!!そもそもスレッドプログラミングって何？って人のために基礎的なお話をしていきます。

そもそもスレッドって何？

スレッドというのは、英語で thread って書きます。意味的には糸ってな意味です。そもそもアプリケーションってのは、それぞれ「プロセス」と呼ばれる単位で動いています。

Windows であれば Ctrl+Alt+Del を押すとタスクマネージャの起動ができるので、それの「プロセス」ってタブをクリックすれば、今現在動いているプロセスを見ることができます。



「アプリケーション」よりはるかに多くの「プロセス」が立ち上がっているのが分かるでしょう？これは見えているプログラムが「アプリケーション」で、見えていないものも表示するのが「プロセス」です。厳密な説明ではないですが、そんなものだと思っておいて下さい。

で、このプロセスというのは1個以上のスレッドで構成されています。自分でプログラムを書く時に特に気にせずに作ると基本的には「シングルスレッド」プロセスが起動します。マルチスレッドの対義語です。

このプロセスの場合、処理は上から下に、自分で書いたコードの順序どおりにただただ実行されます。

例えば

```
int main(){
	cout << "1"<<endl;
	cout << "2"<<endl;
	cout << "3"<<endl;
	cout << "4"<<endl;
	cout << "5"<<endl;
}
```

といったコードを書くと、予想通りに1→2→3→4→5と実行されます。あたりまえです。これがシングルスレッドのコードです。

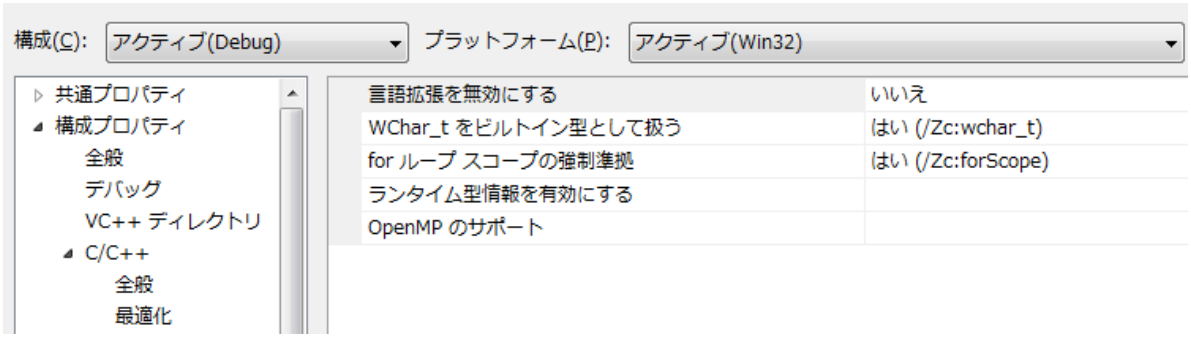
今までのC/C++プログラミングでは、恐らくはこういうシングルスレッドでプログラムを組んでいたのではないでしょうか？

今日はこれをマルチスレッドにしてみようと思います。手っ取り早くやるにはOpenMPを使います。使ってみましょう。

OpenMP初歩入門

まずは

ちなみにWindowsアプリケーションでは同時に複数のプロセスが立ち上がっていますね？そうでないとWindowsの利点がありません。



プロジェクトのプロパティを見て下さい。C/C++→言語→「OpenMPのサポート」ってありますよね？そこを「はい」にしてください。

これでプロジェクトの準備は完了です。あとは、プログラムをいじるだけです。まずはOpenMPの並列化の基本として

#pragma omp parallelを使用します。こいつは、設定したスレッド数だけ、その後のブロックを実行するというやつです。スレッド数を知るにはまず

```
#include<omp.h>
```

をインクルードして

```
cout << "スレッド数="<< omp_get_num_threads() << endl;
cout << "最大使用可能スレッド数="<< omp_get_max_threads() << endl;
```

とでも書けば、現在使用されているスレッド数と、最大使用可能スレッド数がわかると思います。

で、Core2Duoのマシンなら最大スレッドは2。core-i3なら3。core-i5は5 core-i7なら7になるでしょう。イマドキのマシンならショボくても2以上だと思います。

さて、ここまでの時点で現在のスレッド数(omp_get_threads)は1だと思います。では、

#pragma omp parallelでくくってみましょう。

```
#pragma omp parallel
{
```

```
    cout << "スレッド数" << omp_get_num_threads()<<endl;
}
```

どうでしょうか？スレッド数=2と表示されたのではないでしょうか？しかも2つも!!!

既に君はマルチスレッドに片足を突っ込んでいるのだ!!!

さて、その上で先ほどの1～5の出力をくくってみよう。すると

```
スレッド数=1
最大使用可能スレッド数=2
スレッド数=2
スレッド数=2
1
2
3
41
2
3
4
5

5
```

おかしいですね？みんなの所では違った出力になっているとは思いますが、こういう出力になる可能性もあるということです。

本来であれば1122334455だとか1234512345と出るべきだと思います。

…41ってなに？とか、5のあとに改行で5ってなに？

そう、そういう疑問が出てくると思います。こんな感じでマルチスレッドでは奇妙なことが発生します。

なぜか？

図のようにそれぞれのスレッドが勝手に1→2→3→4→5を出力しようとしますが、それぞれのスレッドが自分の都合で出力しますので、Aスレの1→2→3→4まで出力した時点でBスレッドの1→2→3→4→5が出力されており、最後に残ったAの5が出力されているわけです。

これは今までの考え方のままだとかなりキッツいと思います。くれぐれも言っておきますが、それぞれのスレッドは、そのスレッド内の実行順序は守ります。待ったりはしません。

今回の例では最終出力がシングルである標準出力なので、こういう順序になっていますが、

「同時」っていう可能性もあります。

とにかく別スレッドの都合を見て動いたりはしないため、上の出力結果例のように41なんていう出力が発生します。

何故ならば今回の例では

数値出力→改行

という処理ですので、数値出力と改行の間に別のスレッドの処理がかぶるってのは十分にありえるわけです。今回は単なる出力だけですから、大した被害ではありませんが、数値の代入、インクリメント等を行う場合、非常にヤバイ事態が発生します。

今回の例だとそんなにカオスにならないので、ちょっとカオスが発生しやすい状況にしちゃいましょう。カオスにするにはループを使いましょう。更にスレッドも仮想的に増やしちゃいましょう。

ループをマルチスレッドにするには

#pragma omp parallel for を使用します。

さらに、ムリヤリスレッド数を上げるには

omp_set_num_threads(スレッド数);

とします。

例えば図のようにインクリメントを混ぜたとします。

```
int b=0;

int mem[1000];
omp_set_num_threads(7);
#pragma omp parallel for
for(int i=1;i<=1000;++i)
{
        mem[i-1]=++b;
}

for(int i=0;i<1000;++i){
        cout << mem[i] << endl;
}
```

こう書くことにより、ループ分を7分割して、それぞれを別スレッドで代入していきます。

これを書いた人間は mem(i)に順序どおりの値が入っていると思い込んで処理を行うとします。

ただしそうはいきません。通常であれば mem(999)=1000;であるはずですが

このプログラムはマルチスレッドであるため、そうなるとは限りません。

※さらに++bですが、これは32bit 環境であれば大抵の環境においては、アトミック(演算が不分割)であることが保証されているため、ここでは問題ないのですが、アトミック性が保証されていない状態においてはもっと恐ろしい事が起こります…が、ここでは話しません。

アトミック性が保証されていないとbがインクリメントされるタイミングで別スレッドがbをインクリメントし、その結果として、

b=0;

//ここからマルチ(2)

b=b+1;

とした場合、b=2であることが予想されるが、そうとは限らなくなってしまうのだ。そこまで話しちゃうと話し過ぎなので、あとは各自がお仕事についた時に体験しましょう。

ちなみにOpenMPに関しては

http://www.cc.u-tokyo.ac.jp/support/kosyu/03/kosyu-openmp_c.pdf

に良い資料がまとまっているので、興味がある人は見ておきましょう。

Windowsの非同期ファイル操作を知ろう

例えばくっそでけえファイルを読み込みしたいとする

gakuseigamero¥rkawano¥2~3年向け課題¥参考ムービー¥シュー参考ムービー

の中にでけえファイルがあるので、そいつでも使ってくれ。そうすると

```
	cout << "オープン" << endl;

	FILE* fp=fopen("threehundred.avi","rb");//300MBくらいのavi
	fseek( fp, 0, SEEK_END );
	long sz = ftell( fp );
	cout << "ファイルサイズ=" << sz << endl;
	std::vector<char> c(sz);
	fseek( fp, 0, SEEK_SET );
	cout << "読み込み開始" << endl;
	fread(&c[0],sz,1,fp);
	cout << "読み込み終了" << endl;

	fclose(fp);
	cout << "クローズ" << endl;
	getchar();
```

シークにも時間がかかるし読み込みにも時間がかかります。で、この時何が問題なのかというと、fseekって関数やfreadって関数を呼んだ時点で処理が持って行かれてしまうため、アプリケーションはいわゆる「固まる」のである。

自主制作ならそれでいいかもしれないが、製品版だとそうは行かない。これを

…なんかWindows7だとうまくいかないので、ここでの解説はやめにしておきます。実際DxLibの中で非同期ロード関連の関数が有りますが、あれの中身ではファイルロード用のスレッドを一個立ててロードしています。

MSDNのどこにも「が…ダメッ…!」とか書いてないのでキモチワルイんですけどね。まぁ、ここに時間を費やすのは無駄ですから、beginthreadに行っちゃいましょう。

初歩(beginthreadとか)

さて、CreateFile→ReadFileの非同期読み込みがうまくいかないっぽいのであきらめてbeginthreadをやっていきます。

beginthreadってのは、最初に解説した「スレッド」を別に新たに立てることによって、ロード等の処理を別のプログラムにやってもらい、ロード結果だけをうまいこと利用するというやつです。

まずはただただスレッドを立ててみます。

C++でスレッドを立てるには_beginthread を使用します。

マニュアルを見ると…

http://msdn.microsoft.com/ja-jp/library/kdzttdcb(v=vs.90).aspx

なにこれ…

```
uintptr_t _beginthread(
   void( *start_address )( void * ),//別スレッドで実行される関数のアドレス
   unsigned stack_size,//スタックサイズ
   void *arglist //引数リスト(でもなんでもない。たんなる自由なデータポインタ)
);
```

はい。まぁ、深いことは考えずにスレッド作っちゃいましょう。いきなり第一引数がわからない人もいるかもしれませんが、アレですよ。関数ポインタを渡すんですよ。つまり

```
void threadfunc(void*  args){
}
```

こんな感じの関数を作っちゃって第一引数にその関数名を入れてやればいいわけです。とりあえず、1～10000 までの数値を出力するスレッドを立ててみましょう。

関数ができたら main 関数の中で

```
_beginthread(threadfunc,0,NULL);
```

という感じで呼び出して下さい。そうするとひとまずは 1～10 まで出力されると思います。

ちなみに beginthread は process.h に入っているので忘れずにインクルードしましょう。

ここで、_beginthread を実行した直後に、同様な 1～10 の出力処理を書いてみてください。2つの処理がごちゃまぜになって実行されているのがわかると思います。

例えばこのように…

```
threadA=1
threadA=2
threadA=3
threadB=1
threadB=2
threadB=3
threadA=4
threadA=5
threadA=6
threadB=4
```

threadB=5

threadB=6

threadA=7

threadA=8

threadB=7

threadB=8

threadB=9

threadB=10

threadA=9

threadA=10

ねっ？ごちゃまぜでしょ？いくつかの処理は同時発生的に実行されています。このため、理想的な話をすると速度が2倍になるわけです。

…まぁ、それよりも何よりもゲームその他アプリケーションを作る上で重要なのは「画面を停止させない」ということです。さて、というわけでそろそろ本題に入って行きましょう。

どうやっていけば、ロード中に画面を停止させないようにできるのか？簡単です。

「画面の更新」と「ロード」を別スレッドにしてしまえばいい!!!

というわけで作っちゃいましょう。さっきのThreadFunc側にロード処理を。メインスレッド(main関数もしくはWinMain関数に属するスレッドをメインスレッドと呼びます)側に画面更新処理を行う前提でやってみます。

ちなみにC言語のmalloc関数は内部でmutex処理されており「スレッドセーフ」なので別スレッド内部でメモリ確保する際に「かぶったらどうしよう」などという心配はご無用でございます。

…まぁ、こういう「mallocを別スレッド側でやって大丈夫なん？」って思える人のほうが多分あとで伸びてくるとは思います。C/C++ではスレッドセーフでない関数も多いからね。

さて、今回ファイルのロード処理が終わったかどうかは簡単にしたいのでシグナルイベントだのは使わずに素直にポーリング(一定時間で定期的に状況をチェックすること)します。

http://ja.wikipedia.org/wiki/%E3%83%9D%E3%83%BC%E3%83%AA%E3%83%B3%E3%82%B0_(%E6%83%85%E5%A0%B1)

ファイルサイズの計算はfstatとか使ったりGetFileSizeExを使ってもいいんですが、fopen系だけで行きたいなら

fseekとftellを使用します。fseekはシークね。シークバーのシークですよ。ニコニコ動画とかでもあるでしょ？で、

```
fseek(fp,0,SEEK_END);
```

ってやるとファイルの最後(ニコ動で言うと動画の最後で、提供だの宣伝だの出てくる場所)

fseek(fp,0,SEEK_SET);

ってやるとファイルの先頭(ニコ動で言うと動画の頭ですね)

になります。で、

ftellの役割は、現在のファイルの場所(ニコ動で言うと現在のシークバーの位置が何分何秒なのかって情報)を返します。

つまりこれを使用してファイルサイズを知るにはファイルの末尾に行って現在位置を取得し、次にファイルの先頭に行って現在位置を取得してから、その差を計算すればいいわけです。

つまりこう

```
	//ファイルサイズ計算
	fseek(fp,0,SEEK_END);
	int tailpos=ftell(fp);
	fseek(fp,0,SEEK_SET);
	int filesize=tailpos-ftell(fp);
```

で、このファイルサイズを用いてmallocだのnewだのでメモリの確保したあとでfreadでファイルのロードを行います。

一連の処理が終わったら、終わったことが分かるように引数で渡された部分にフラグを割り当てておき、そこに「終わったよー」をしらせればいい。

```
 #include<iostream>
#include<vector>
#include<Windows.h>
#include<process.h>
using namespace std;

///ファイルロード用構造体
struct LoadState{
	void* data;//データポインタ
	bool endread;//読み終わりフラグ
};

///読み込みスレッド関数
///@param args voidポインタで渡されるのでLoadState型にキャストして使う
void ThreadFunc(void* args){
	FILE* fp = fopen("UnitySetup-4.3.exe","rb");

	LoadState* ls=(LoadState*)(args);

	//ファイルサイズ計算
	fseek(fp,0,SEEK_END);
	int tailpos=ftell(fp);
	fseek(fp,0,SEEK_SET);
	int filesize=tailpos-ftell(fp);

	//必要なメモリを確保
	ls->data=malloc(filesize);//malloc関数はスレッドセーフ
	//いよいよ読み込みでござる。
```

```
        fread(ls->data,filesize,1,fp);
        fclose(fp);
        ls->endread=true;
}

int main(){
        LoadState st={};
        st.data=NULL;
        //ロードスレッド起動
        _beginthread(ThreadFunc,0,&st);
        int count=0;
        //ロード完了待ち
        while(!st.endread){
                cout<<"count"<< ++count << endl;
                Sleep(100);
        }
        cout<<"read end" << endl;
        char s[4]={};
        memcpy(s,st.data,3);
        //読み込んだ証拠に最初の３文字を出力
        cout<<"data of head3="  << s <<  endl;

        getchar();
}
```

ワーカースレッドつくろー

スレッドプールつくろー

コラムです

課題：データ構造とアルゴリズム基礎

期限:来週(10/17)まで
gakuseigame¥ゲーム¥GC 専 3¥C++課題提出先¥１０月第 2～第 3 週課題
に自分の名前のフォルダを作って、そこに提出して下さい。

今回の課題は、九州の某会社にて出題された課題の一部と同等のものです。割と基礎的な課題を伝統的に出題している会社も多いので、こういう物を解いておくといいかと思います。

課題1:リスト構造

リスト構造は1年の時にやっていると思いますので、あくまでも復習ということで。まぁ、スキンメッシュアニメーションやる人はリストより難しいツリーやらなきゃいけないので、予習のための肩慣らしだとも言えます。

リスト構造というのは、メモリ上では本来バラバラであるデータを連結させるためのアルゴリズムです。

```
/*****************************************************************************************/
/**
   問題【リスト】

   以下の構造体 struct _data について、value の値が小さい順に追加されていくというプログラムを作りたい。
   以下のコードに示すように、データ追加に使用する DataAdd という関数を実装せよ
   データ塊が空の場合(つまり、*top が NULL)も考慮せよ。
   コマンドラインから入力されたデータから、DataAdd を使用し、入力が数値であればリストに追加していき、そうでなければリストを小さい順に表示する処理を実装せよ。
   ※以下のコード以外の#include は使わないように。
   ※ 参考：データ塊のイメージ

   *top             d1                    d2                    d3
   +-------+        +--------------+      +--------------+      +---------------+
   | d1    | -----> | next  = d2   | ---> | next  = d3   | ---> | next  = NULL  |
   +-------+        +--------------+      +--------------+      +---------------+
                    | value = 3    |      | value = 6    |      | value = 10    |
                    +--------------+      +--------------+      +---------------+

*/
/*****************************************************************************************/
#include <stdio.h>
#include <stdlib.h>


typedef struct _data{
     struct _data *next;  /* 次の構造体へのポインタ(NULL が終了) */
     int value;           /* 値(この値の小さい順にチェーンされる) */
} DATA;


/* ========================================================================== */
```

```
/**
   @brief
   データ塊にデータを付け加える
   @param top
   データ塊の始まりを格納したワークへのポインタ
   @param add
   付け加えるデータ(value 以外は不定値である)
*/
/* ============================================================================== */
void DataAdd(DATA **top,DATA *add)
{
}
//================================================================================
//ここから DataAdd 関数を呼び出し scanf なりなんなりで入力されたデータを追加していき、
//数値以外が入力された時に整列済みデータを出力する処理を記述しなさい
//================================================================================
int main(){
     //
}
```

解説：リスト構造

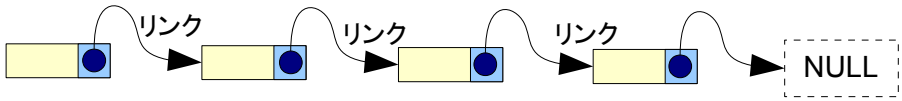
はい、この課題はいかがでしたか？うん、たぶんこれに1週間かけてできなかった人は基本情報技術者試験もツラかったんじゃないかな？
何しろC言語の「アルゴリズムとデータ構造」の中で一番基礎的なものですから。これができなければ、たいていの問題には苦労するでしょう。

あと、誰とは言いませんが、プログラムが終了しないように無限ループを使用していたツワモノが居ます。C言語のコマンドラインプログラムにて、待ちを使いたいなら。
プログラムの最後でgetchar();と書くことをお勧めします。

```
while(true){
	continue;
}
```

なんて書いちゃうと、CPU時間食いまくりますので、やめていただきたい。いや真面目な話。

さて、閑話休題。リスト構造ですが、リストってのは「次のデータの場所」を持っておくことによって芋づる式にデータを組んでいくって構造です。



みたまんま芋づるですね。
で、C言語において、大抵の場合は「次のデータの場所」をポインタで持っておきます。つまり次のデータのアドレスを知っとく必要があるわけですね。

で、この時に使うのが「**自己参照構造体**」です。といっても名前はどうでもいい。覚えなくてもイイですよ。
どういうものかというと、自分のメンバ変数の中に、自分の型を指し示すポインタ型の変数を持っておけばいい。
つまり、

```
struct List{
	int value;//値
	List* next;//次の場所
};
```

これだけでいいのだ。簡単だろう？ええっ？なんでListの型自体が確定していないのにそいつのポインタを使えるのかって？
ハハハッ!!ワロス。
だって、型のサイズは確定するでしょ？コンパイラ的にはそれだけで十分なのさ。
型Listのサイズはいくつだよ？

そう、intで4バイト、List*で4バイト。8バイト。つまりこいつの型自体が8バイト食うことが分かっている。そこまでわかっていれば、ポインタを使うことができるのだ。

もう少し先でお話するが、「前方参照」を使えば別に中身がわかってなくてもポインタを使うことが可能である。これについてはもうちょっとあとで話す。とにかく今はリストだ。
ここまで知っていれば、問題文にわざわざ

```
typedef struct _data{
     struct _data *next;  /* 次の構造体へのポインタ(NULL が終了) */
     int value;           /* 値(この値の小さい順にチェーンされる) */
} DATA;
```

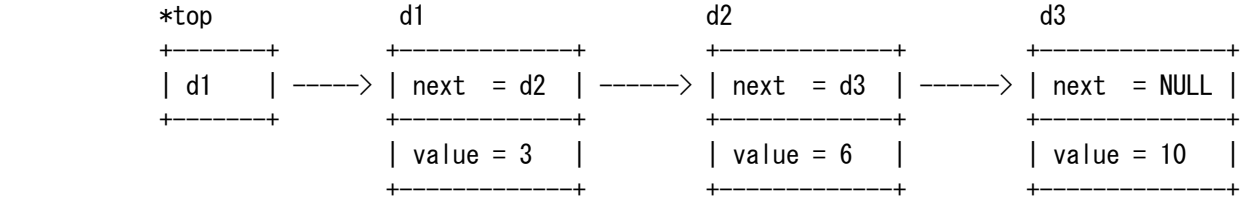
とかって書いてなくても自分でリスト構造体を書くことだってできる。分かっている人間からすれば、この問題は「ウッハーＷＷＷ、親切親切ぅ!!」
と思えるはずなんですよ。

はい、更に親切なことには関数の枠まで作ってくれております。

```
void DataAdd(DATA **top,DATA *add)
{
}
```

こうでございますので、あとは仕様通りに作ればいいのですよ。フツーにリスト構造の基礎がわかってればできるでしょ。
ゲーム会社の仕様なんざこんなやさしくないで？ていうか仕様なんかないんやで？
仕様ってのは

以下のコードに示すように、データ追加に使用するDataAddという関数を実装せよ
データ塊が空の場合(つまり、*topがNULL)も考慮せよ。
コマンドラインから入力されたデータから、DataAddを使用し、入力が数値であればリストに追加していき、そうでなければリストを小さい順に表示する処理を実装せよ。
※以下のコード以外の#includeは使わないように。
※ 参考：データ塊のイメージ



これのことね。
ここさえ見ればあとは注文通りプログラムするだけ。これができないならゲーム会社でゲームなんか作れへんよ？

どうしよっかね？引数のaddがNULLってのは考慮しなくてもいいっぽい(考慮してもいいよ)ので、addはきちんとオブジェクトが入っていると仮定して話を進めます。

```
if(*top==NULL){//topがNULLってのは何もデータがないのでaddつっこんで後は関数抜ければいい。
     *top=add;
     return;
}
```

で、topが入っている(NULLではない)場合は、topの後につなげればいい。簡単だね？

```
(*top)->next=add;
```

はい、これだけでつながりました。簡単ですね。ただし、これをやっていいのはnextがNULLの時だけなので、

```
if((*top)->next==NULL){
	(*top)->next=add;
}else{
	//…何を書けばいいんだろうか？
}
```

と書きます。nextに元々値が入っていればNULLではない。値が入っているところをaddにしてしまうと、前のが上書きされてしまうので、以前のnextにアクセスする術がなくなってしまう。

どないすりゃええんすかね？
あー、nextもListやねんから、nextのnextに書いたらエエんちゃうん？そうだねプロテインだね。

```
if((*top)->next==NULL){
	(*top)->next=add;
}else{
	(*top)->next->next=add;
}
```

こうですねわかります。
これでいいんですか？これでいいんですかぁ？良くないですよね。だって、その次のnextもNULLじゃなかったらどうするんですか？

```
if((*top)->next==NULL){
	(*top)->next=add;
}else if((*top)->next->next==NULL){
	(*top)->next->next=add;
}else if((*top)->next->next->next==NULL){
	(*top)->next->next->next=add;
}else{
	…
}
```

はい、おわかりのようにやってられませんね？いくつ繋がるのかなんてわからないんですから…
じゃあどうすんのさ？

NULLに到達するまで探し続けるのさ。**「非NULLなら、NULLまで下ろうリスト構造」**です。

何のためにwhile文があると思ってるんだ？別に無限ループ作るためにあるんじゃないんだぜ？

全てがList型なんだから、現在の最下層を示す一次変数tmpを用意しておく。で
既にtopがNULLだったらばreturnしてるとすると。

```
List* tmp=*top;
while(tmp->next!=NULL){//NULLになるまで降りていこう
    tmp=tmp->next;
}
//ここに来た時点でtmp->nextはNULLであるはずなので、そこに放り込んでやる。
tmp->next=add;
```

おわり。簡単でしょ？とりあえずリストへの追加だけを考えるなら、これで終わり…。まぁいいし。
今度はmain関数を考えようか。

```
//ここからDataAdd関数を呼び出しscanfなりなんなりで入力されたデータを追加していき、
//数値以外が入力された時に整列済みデータを出力する処理を記述しなさい
```

○数字以外が入力されるまで、入力を待ち続ける。つまり数字が入力されたら次を待つ状態にならなければならないし。
○scanfかなんかで入力を受けとる。

これだけです。逆に言うとこれだけあれば十分っていうことですね。

あれ…待ち続けるってのがわからない？えっ？ナニソレ怖い。

```
DATA* top=NULL;
while(1){
    //数値解析処理
        //数値じゃなかったらループ抜ける
    //追加処理
    DATA* add=new DATA();
    add->value=入力された数値;
    add->next=NULL;
    DataAdd(&top,add);
}
```

はい、数値解析処理ですが、scanf関数は読み取りに成功した要素の個数を返します。じゃあ、失敗したらどうなるのかというと、0を返します。
まぁ、基本的には1が返ってきたら成功で、0が返ってきたら失敗ということでいいでしょう。そうなると例えば

```
int in;
int ret =scanf("%d",&in);
```

とでもしてやるとretに結果が入っていますので
if(ret==1)の時にDataAdd処理をすればいいだろう？

まとめて書くとこんな感じ？

```
	int ret = 0;
	int in;
	DATA* top=NULL;
	do{
		ret=scanf("%d",&in);
		if(ret==1){
			DATA* add=new DATA();
			add->value=in;
			add->next=NULL;
			DataAdd(&top,add);
		}
	}while(ret!=0);
```

たぶんこれでドゥンドゥン入っていきますわな？
じゃあ、あとはこれを全部表示してみましょう。
配列じゃないので、単純な for 文じゃ無理だ。

てっぺんから芋づる式に見ていくのがリスト構造のやり方だ。一般的な教科書的に書くなら、こう。

```
	DATA* current=top;
	while(current!=NULL){
		current=current->next;
	}
```

リスト構造がきっちりできていれば、next メンバが次の要素を指し示すので、これが NULL になるまで繰り返せばいい。
あとは、next になる前に出力コードを書けばいいので

```
while(current!=NULL){
	printf("%d,",current->value);
	current=current->next;
}
```

とでも書いておく。まとめると…こう

```
#include<stdio.h>
#include<stdlib.h>

typedef struct _data{
	int value;
	struct _data* next;
} DATA;

void DataAdd(DATA** top,DATA* add){
	if(*top==NULL){
		*top=add;
		return;
	}
	DATA* tmp=*top;
	while(tmp->next!=NULL){
		tmp=tmp->next;
	}
```

```
        tmp->next=add;
}

int main(){
        int ret = 0;
        int in;
        DATA* top=NULL;
        do{
                ret=scanf("%d",&in);
                if(ret==1){
                        DATA* add=new DATA();
                        add->value=in;
                        add->next=NULL;
                        DataAdd(&top,add);
                }
        }while(ret!=0);
        printf("データっ！出力っ！！");

        DATA* current=top;
        while(current!=NULL){
                printf("%d,",current->value);
                current=current->next;
        }
        char c;
        do{
                c=getchar();
        }while(c!='e');
}
```

だけど、何か忘れているよね？そう

リストを小さい順に表示する処理を実装

これが抜けている。これね、簡単なのよ。よ～く考えてね。
別に入力した順に追加していけとは書いてない。最終的に小さい順に表示すればいい。ということは、DataAdd の時点で調整してやればソートなんか必要ないってわけ。

NULL になるまで下っていくところでちょっと細工を入れておく。

```
        while(tmp->next!=NULL){
                tmp=tmp->next;
        }
        tmp->next=add;
```

この部分ね。例えば next にモノがある場合であっても、入れようとしている値が次のものより小さいなら間に入れるべきです。
そもそもこの「間に入れる」ってのを簡便にするためのものがリスト構造なわけで…

```
while(tmp->next!=NULL){
        if(add->value<tmp->next->value){
                DATA* swap=tmp->next;
```

```
            tmp->next=add;
            add->next=swap;
            return;
        }
        tmp=tmp->next;
    }
```

これでいいのかな？本当にそうかな？そう、このままでは先頭だけがそのままになっちまう。top もすげ替えられるようにしなければならない。
つまり、*top->value と add->value も比較できなければならない。

ループ前にこう書いとく

```
    DATA* tmp=*top;
    if(add->value < tmp->value){
            *top=add;
            add->next=tmp;
            return;
    }
```

全てまとめると、こう。

```
#include<stdio.h>
#include<stdlib.h>

typedef struct _data{
    int value;
    struct _data* next;
} DATA;

void DataAdd(DATA** top,DATA* add){
    if(*top==NULL){
            *top=add;
            return;
    }
    DATA* tmp=*top;
    if(add->value < tmp->value){
            *top=add;
            add->next=tmp;
            return;
    }
    while(tmp->next!=NULL){

            if(add->value < tmp->next->value){
                    DATA* swap=tmp->next;
                    tmp->next=add;
                    add->next=swap;
                    return;
            }
            tmp=tmp->next;
    }
    tmp->next=add;
```

```
}

int main(){
	int ret = 0;
	int in;
	DATA* top=NULL;
	do{
		ret=scanf("%d",&in);
		if(ret==1){
			DATA* add=new DATA();
			add->value=in;
			add->next=NULL;
			DataAdd(&top,add);
		}
	}while(ret!=0);
	printf("データっ！出力っ！！");

	DATA* current=top;
	while(current!=NULL){
		printf("%d,",current->value);
		current=current->next;
	}
	char c;
	do{
		c=getchar();
	}while(c!='e');
}
```

最終課題

ここからの課題は全てコマンドラインアプリです。なお、誰の制作物であるかを証明するために、起動時に1行目に

学籍番号_氏名

と全てのプログラムで出力できるようにしておいて下さい。

なお、提出期限は

1月16日(木)です。

※実装は問いませんが、C++の機能を利用しているのが望ましいです。

gakuseigame¥ゲーム¥GC専3¥C++課題提出先¥最終課題提出先

にその1その2その3てありますので、そこに自分の学籍番号と氏名がわかるように提出して下さい。

その1

コマンドラインから文字列を入力させて、最終的に連結文字を出力する。
例:

僕と(Enter)
契約して(Enter)
魔法少女になってよ(Enter)
(Enter)
で、これらの文字を連結してコマンドラインに

「僕と契約して魔法少女になってよ」

と出力させてみてください。

もちろん、別の文字でもやれるようにしなければダメです。
例2:
これから(Enter)
毎日(Enter)
家を(Enter)
焼こうぜ(Enter)
(Enter)
では

「これから毎日家を焼こうぜ」

と出力できるようにお願いします。

その2

2次元ベクトルの演算を行って下さい。
連続して二次元ベクトルをコマンドラインから入力しますので、それの和、差、内積、外積を出力できるようなプログラムを作って下さい。
例:
1,2(Enter)
3,4(Enter)
と入力されたら

和:4,6
差:-2,-2
内積:11
外積:-2

と出力できるようにしておいてください。
もちろん、様々な値に対応できるようにしておいて下さい。

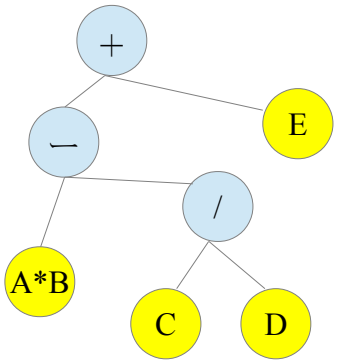
その3

ポーランド記法で入力された演算文字列を計算し、その結果を出力して下さい。ポーランド記法とは
A+B+Cという式であれば"+ + A B"という文字列であり
A*B-Cという式であれば"- * A B"という文字列であり

A*B-C/D+Eという式であれば"＋ － ＊ Ａ Ｂ / Ｃ Ｄ Ｅ"である。
というような式である。

もし成り立たない式が入力された場合はエラーを表す文字列を表示して下さい。

ヒント:こいつはスタックを使うとラクに実装できます。「演算」と「値」を分けて考えて、「演算」は後の「値」を2つ受け取り演算する。
演算の結果はまた「値」となる。
ここまでを「理解」できれば、…まぁ作れるんじゃないかな。頑張ってみてね。ツリーでいうと、演算子は2つの「子」を持っており、「子」は「演算子」または「値」である。「値」であるのならばリーフになる。
つまり演算子はノード、値はリーフである。
つまり上の3番目の例で言うと＋が来た時点で後の2文字を調べる。後にーがあるためこいつも2つの子をとる。＊がーの子ノードとなる。＊の子はABなのでAとBをかけて値になる。この時点で値A*Bはリーフである。とすると後に控えている/が2つのC,Dをとり、こいつらはリーフとなるため、+のもう一つのリーフはEとなる。



もちろんC/Dもリーフ扱いとなる。
とりあえず、ツリーが成り立たない(末端がリーフでない)場合はエラーを出力するようにして下さい。