Learning Object-Oriented Programming of C++

このテキストは紅白の巫女さんと黒白のゴスロリさんによって書かれました[i]

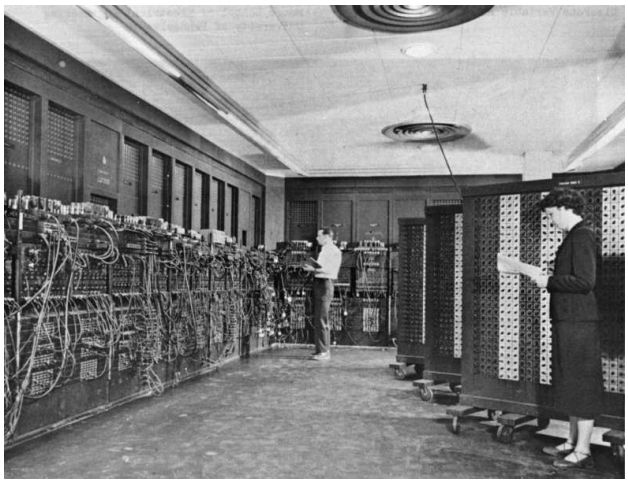
C++言語を用いてオブジェクト指向プログラミングの初歩の解説をします。対象者は基本的な C 言語が扱える者[ii]でオブジェクト指向？　何それおいしいの？　というレベルの人です。ちなみに表題の Object-Oriented がオブジェクト指向という意味らしいです。

第一章　オブジェクト指向とは

何事もまずは全体把握や用語解説から入るのが鉄板なのでそれに習ってソースコードではない方面からオブジェクト指向とは何かを説明します。

第一節　プログラミングの抽象化

プログラミングの容易さの歴史というものは抽象化のことをさします。世界最初のコンピュータ[iii]と言われている ENIAC[iv]は大量の真空管をケーブルを使って相互に接続し加算、減算などの処理をしていたそうです。ここでのケーブルの接続というのは文字通りオーディオの配線をするようなものだったらしいです。ここでのプログラミングというのはケーブルの接続とイコールになります。今のソフトウェアとはかけ離れた物理的なプログラミングというわけです。しかしプログラミングするには非常に手間がかかり問題点は明白です。ENIAC の後継の EDVAC というのが開発されます。EDVAC の特徴はプログラム内蔵方式を採用したことです。プログラム内蔵方式とはプログラムを中のメモリに保存し CPU がそのメモリの中身に沿って演算をする方式のことでつまりは今のコンピュータと同じ方式です。20 世紀最高の天才のフォンノイマン[v]によって提唱されたコンピュータアーキテクチャです[vi]。



世界最初(?)のコンピュータ ENIAC

ここでのプログラミングというものはメモリ上に 2 進数をおくことであって先のケーブル配線によるプログラミングとは大きく変わってきます。物理的電気回路からソフトウェアへと一歩抽象化されたことになります。

まだ 2 進数をおく時代です。この 2 進数の羅列を機械語と人間は名付けます。さすがに当時の

---

i　脚注の権限は私がもらった

ii　関数までは使えるようになっていて欲しい

iii　コンピュータというものが何かという議論になるので深くはツッコミは入れないように

iv　1946 年アメリカで弾道計算の目的として作られた

v　1903-1957、ハンガリーの数学者。一言でいうと変態。コンピュータより計算が速い頭脳の持ち主(比喩ではない)

vi　この辺の詳しいことは情報科の人に任せる。

人も 0 と 1 の数字の群を扱うのは嫌になったのでしょう、命令ごとに名前を名付けそれらでプログラミングをするようになります。例えば足し算を「ADD」と名付けるようなことです。そのような名前でプログラミングをし、あとで一括でそれらの単語を 2 進数に変換するという手法がとられるようになりました。アセンブリ言語[i]と呼ばれるものです。このアセンブリ言語は機械語とそう手間は変わらないにもかかわらず 21 世紀に入った現代でも要所要所で使われています。組み込みマイコンやコンピュータの BIOS などに今も根強く残りホントに一番最下層の重要なところで使われている言語です[ii]。

一番初期の UNIX もアセンブリ言語で記述されていましたがすぐに皆さんお馴染みの C 言語で書き換えられることになりました。C 言語は UNIX を記述するために開発され、「MOV」なんていう命令よりも「=」という記号を使って自由に数式を記述できたほうが楽ではないかという思想から誕生しました。ここでまた一つ抽象化されています。CPU の命令語とは全く無関係のものを記述してそれを機械語に変換する[iii]という形へと変化しました。抽象化することによってより人間にわかりやすく、物理的なものから離れソフトウェア的なものに進化していることがわかります。C 言語ではある一定のまとまりを関数という形で一つの物としそれらを組み合わせていくことによってプログラミングをするという手法が用いられてきました。

さてここでオブジェクト指向の登場です。オブジェクト指向というのは対象物に着目してそれらについて記述していくプログラミング方法のことです。例えばデスクトップのアイコンがあったとします。そのアイコンを表示するプログラムの場合、アイコンの座標値や実際の描画方法などを記述します。従来の関数によるプログラミング方法は関数の組み合わせであって具体的な物との結びつきがありませんでした。しかしオブジェクト指向というのは対象する物を中心として考え、それに機能を追加していくという形へと変化しました。

オブジェクト指向プログラミングをしたことがない人には実感がわかないと思いますがプログラミングの抽象化の歴史からオブジェクト指向という発想が生まれたということを理解してください。

第二節　オブジェクト

オブジェクト指向と聞いたときにオブジェクトとは何ぞやというのが正直なところだと思います。英和辞典を引いたところで物とか物体という意味しか出てきません[iv]。ここでいうオブジェ

---

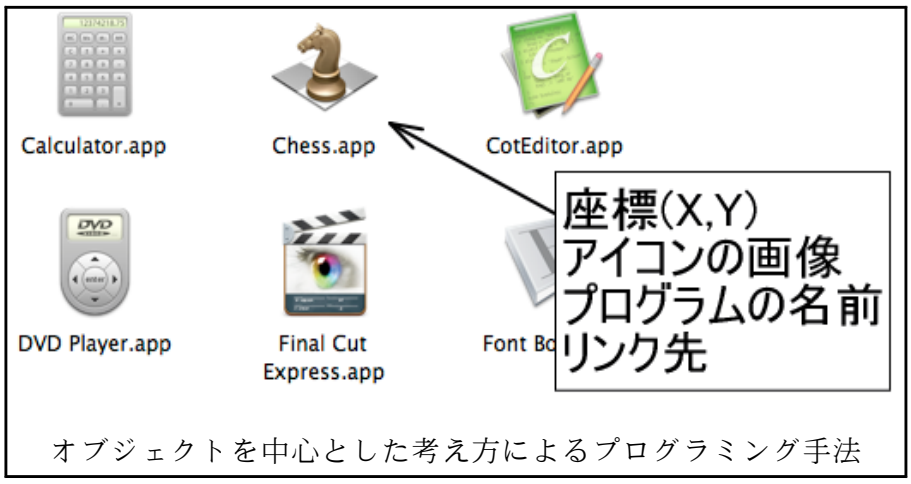
i　機械語に変換することが「アセンブラ」です。アセンブラ勉強しろとよく言われますがそれはアセンブリ言語を勉強しろという意味です

ii　アセンブリの利点は人間が最終結果に近いコードを書くため最適化ができる点。また最適化がすさまじくできるため処理能力が低い処理系でも使われることが多い(ファミコンやスーファミなど)。かなり逸れるが、ファイナルファンタジー 3 はファミコンという限られたリソース(ROM 容量 4Mbit)の中であの壮大な物語が作られた。開発にはナーシャジベリというプログラマがいた。まさに変態と言えるプログラム技術によって完成したと言われている。あまりにも最適化が激しかったためか常人(普通のプログラマ)ではプログラムを読むことができず、FF3 のリメイクまでに 16 年の歳月を要した。ナーシャのゲームを見た任天堂社員は自社のハードでこんなことができるはずがないと言った。ちなみにファミコン時代の FF のスタッフロールに一番最初に出てくる名前はナーシャである。さらにナーシャはファミコンなのに 3D スクロールゲームも開発した。はっきり言って訳がわからない

iii　機械語に変換するのをコンパイラといいます。実際のところは C 言語などの高級言語は一旦アセンブリ言語に変換され、そのアセンブリ言語を機械語に変換するという作業をしている

iv　実際に調べてみたら物、物体、対象、的、目的、目当てとかとかでてきた

クトというのは一つひとつのデータのことです。データというのは Excel のセルの各々だったり iTunes の曲の一曲だったりゲームの自機や敵キャラのことを指します。それらオブジェクトにはそのオブジェクトの必要な情報が入っていることでしょう。先の例ならばセルの値や曲名、曲の長さのことです。もっとプログラミング的な言い方をするならば変数で構成された塊、すなわち構造体のことをオブジェクトと言うこともできます[i]。



オブジェクトを中心とした考え方によるプログラミング手法

第三節　オブジェクト指向言語

最近のほとんどの言語でオブジェクト指向が使えます。現在使われている言語でオブジェクト指向をサポートしていないのは C 言語くらいです。オブジェクト指向言語として有名なものとして Smalltalk が上げられます。聞き慣れない言語ですがオブジェクト指向言語としてはかなり大きな影響を与えた言語で C++や Java は Smalltalk に影響を受けたと言われています。現在はあまり使われておりませんが後継の Objective-C で Smalltalk のオブジェクト指向っぽさが体感できます。

今回このテキストで行う C++は C 言語にオブジェクト指向を拡張した形なので完全なオブジェクト指向言語とは言い切れない側面があります。何故かというと C 言語を拡張したためにオブジェクト指向言語としては扱いにくい箇所が多々あります。ですが他の言語に比べてかなりの高速動作ができゲームなどリアルタイム処理が必要なところでは C++が非常に重要なところを占めています[ii]。一方の Objective-C は C 言語とオブジェクト指向が両立している形なのでオブジェクト指向の記述部は C 言語とは全く別の側面を持った言語として記述できます[iii]。Objective-C はインラインの C とも呼ばれそれぞれのオブジェクト間が動的に結ばれるといった他の言語にはない柔軟性を持っています。Windows で使う機会は非常に希ですが、NextStep や Mac　OSX で使われています。最近は iPhone の普及に伴い注目を浴びている言語でもあります。せっかくなので Objective-C のプログラムの例をあげます。

```
1  #include<stdio.h>
2  #import<objc/Object.h>
3
4  @interface superClass:Object{
```

i　実際 C++は構造体を拡張する形でオブジェクト指向を実現している

ii　もっと実行速度が求められる分野ではアセンブリが使われる。なお高級言語で実行速度最高は Fortran

iii　C++はオブジェクト指向が混ざっているのに対し、Objective-C は完全に別のような言語がもう一つあるような感じ

```
5       int x;
6   }
7   -(void)method;
8   @end
9
10  @implementation superClass
11  -(void)method{
12      printf("superClass.method¥n");
13  }
14  @end
15
16  int main(void){
17      [[subClass alloc] method];
18      return 0;
19  }
```

C 言語との互換性とオブジェクト指向を両立させた言語として、C++とは違ったクラスの実現を行っています。一方で printf を初めとする関数といったものも使える言語として水と油のような混ざっていない言語というのが特徴です。

また Sun が開発した Java[i] もオブジェクト指向言語です。Java は全てオブジェクト指向で記述するという形のプログラミング方式です。また Java の影響を受けた C#[ii] も同じ構造となっています。main 関数から何から何まで全てがオブジェクト指向のクラスというものに属し綺麗に整頓された言語として使われています。話は逸れますが Java はオブジェクト指向という面も強いですがそれ以上に処理系に依存しない実行コードという面も持っています。今までのプログラミング言語は最終的には機械語にまで翻訳されて完成という形を取っていましたが、Java は中間コードで完成という形をとっています。動作させるにはランタイムが必要ですが逆にランタイムさえあればどんな処理系でも動作可能という強みを持っています。C#も同じような方式をとりマイクロソフトは.net と呼ばれる方式に移行しつつあります。すなわち OS に依存しない実行ファイルというのを目指しているわけです。これは現在のコンピュータが非常に高速になったため逐一機械語に翻訳するという形でも十分な動作が可能となりより、将来はモバイル機器でもデスクトップ PC と同じようなソフトウェアが使えるようになります[iii]。

様々なオブジェクト指向言語

第四節　クラスとインスタンス

プログラム内においてオブジェクトはクラスという形で記述されます。C++の場合、クラスと

i　ジャワコーヒーと同じ綴りだけど PC の Java は「ジャバ」と発音する。あと与太話だが Objective-C の API は Cocoa という。ようはコーヒーときたからにはココアというわけだ。

ii　マイクロソフトが作った Java のパクり言語。だけどすごい

iii　モバイル機器と普通の PC では CPU の種類が違うため機械語に変換した時点でその対象機器しか動かなくなる

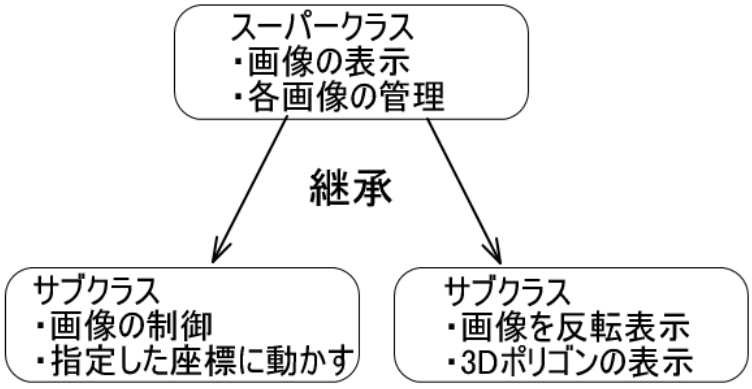
いうのは構造体を拡張した形として記述され構造体の記述方法とほぼ同じ方法をとっています。クラスというのはオブジェクトの変数やメソッド(後述)の集まりのことをクラスと言います。つまりクラスというのはオブジェクトの型[i]に相当するものととらえてください。

先ほど出てきたメソッドという言葉についても説明します。メソッドというのはクラスの機能を記述したコードのことを指します。C 言語風に言うならば関数です。記述の仕方自体は C 言語の関数とほぼ同じですが、厳密にはメソッドと関数は違う意味合いを持ちます。メソッドはクラスの内部のオブジェクトの機能をまとめるもので、関数というのは単独で存在するものです。メソッド≒関数と覚えておいてください。

インスタンスというのはオブジェクトの実態を指します。先のクラスが型に相当するならばインスタンスは型を使ってできた具体的な物です。一つのクラスに対し、インスタンスは複数個作ることができます。例えば敵キャラというクラスを作ります。敵キャラクラスには座標やヒットポイントの変数、敵の移動や攻撃のメソッドなどが含まれています。これらの情報をもとに実際に画面に出てくる敵キャラをインスタンスと呼びます。画面に出てくる複数の敵キャラは一つひとつに座標値やヒットポイントが違うはずです。これらは別個にインスタンスが作られているため同じクラスでも別々の値を持つことができます。

インスタンスが持つ変数のことをインスタンス変数と呼びます。またインスタンスが持つメソッドのことをインスタンスメソッドと呼びます。対になる言葉としてクラス変数、クラスメソッドというものが存在します。インスタンス変数がインスタンス一つひとつが持つ変数に対して、クラス変数というのはクラスにただ一つしか持たない変数のことです。上記の敵キャラの例で言うならば敵キャラの初期 HP や画像のファイルはクラス変数を使用することができます。初期 HP や画像ファイルは同じ種類の敵キャラならば全て同じ値や画像を使用できるためクラス変数として使うことができます。クラスメソッドも同じようにクラス全体の処理をさせるためのメソッドです。クラス変数の値を変更するときなどに使えます。

第五節　継承



クラスは大抵は一つのファイルに一つのクラスを記述します[ii]。それは何故かというと可搬性をよくするためです。以前記述したコードは再利用しようという考えで、結果として作るソフトが違ったとしても似たようなオブジェクトは沢山あるはずです。例えばキャラクタ画像を表示す

i　オブジェクト指向言語では大抵はオブジェクト型というのがある(C++は何故かない)

ii　ヘッダがあるので 1 つのクラスに実質 2 つのファイルになってしまうが

るというクラスがあったとします。それは STG だろうが ACT だろうが 2D ゲームであれば基本的に使い回すことができます。

しかし、一部の機能だけを付け足したかったり、この機能だけ動作を変えたいと言った場合があります。そのときに使うのが継承というオブジェクト指向の機能の一つです。継承というのは元となるクラスから機能を拡張したり、機能を書き換えたりすることができます。元となるクラスのことを基底クラス、親クラス、スーパークラスなどと言います。逆に継承したクラスのことを派生クラス、子クラス、サブクラスなどと言います[i]。派生クラスは基底クラスの機能、つまり変数とメソッドを全て受けつくことができ、さらに追加でメソッドや変数を追加することができます。また既に定義されているメソッドを書き換えることもできます。

また仕様変更など関数、メソッドに変更が生じたとき、継承を用いるとごく一部の部分だけを書き換えることによって派生クラスを含めた仕様変更ができるため非常に開発効率を高めることができます。

i　いろんな呼び方があるんだけどニュアンスでわかると思う

第二章　オブジェクト指向に入る前の C++

気難しい用語の説明は書いてるのも飽きてきたのでソースコードの説明を書きます。が、いくつかおさらい事項とせっかくの C++を勉強するので Hello, World!!のソースコードを示します。

第一節　Hello, World!!

まず C++の拡張子は.cpp です。C 言語が.c に対して.cpp なのでなんとも行儀がいいです[i]。C 言語と C++は基本的に上位互換なので C 言語で使えたことはほぼ全て使えます[ii]。もちろん printf 関数も使えます[iii]。なので printf("Hello, World!!");と書けばコンソール画面にはきちんと「Hello, World!!」と表示されます。ですが、それだと新しい言語を始めた感がない[iv]ので C++で Hello, World!!を記述します。

```
1  #include<iostream>
2  using namespace std;
3
4  int main(void){
5      cout << "Hello, World!!" << endl;
6      return 0;
7  }
```

ちょこっと C 言語とは変わりました。「Hello, World!!」と表示するというのはわかると思いますが、「<<」この記号でなんかやってるし……　というのが見た感想だと思います。これの意味を理解するのは結構後のほうだと思うのですが、ここでは C++は C 言語とは違う言語というのを理解してください。

printf の%d に対応するものは次のように記述します。

```
1  #include<iostream>
2  using namespace std;
3
4  int main(void){
5      int num = 10;
6      cout << "numの値は" << num << endl;
7      return 0;
8  }
```

ここでは変数を int 型を用いましたが、double 型や char 型であってもかまいません。printf では%d や%f などコードで記述しなければなりませんでしたが、cout では自動的に型の判断をしてくれます。またこの「<<」記号、本来は左シフト演算子と呼ばれるものなのですが、この iostream では cout に流し込む演算子として機能しています[v]。

第二節　変数の宣言

C 言語と C++の違う点も細かい点ながらもいくつかあります。例えば C++言語では次のようなソースコードを頻繁に書くようになります。

---

i　ヘッダに関しては.h で C も C++も変わらない。ちなみに Java は.java、Perl は.pl、Objective-C は.m。

ii　厳密に書くのって難しいね。「ほぼ」ってなんなんだろうね

iii　ちなみに筆者は C++でも printf を使う

iv　新しい言語を始めたら最初にやることは必ず「Hello, World!!」と表示させること。異論は認めない

v　ようするに演算子のオーバーロード

```
1   int main(void){
2       for( int i = 0; i < 10; i++){
3           //なんかの処理
4       }
5   }
```

for 文の括弧の中での変数宣言をすることができます。地味ですがとても便利です。特に関数が長くなってくると for 文で回すための変数を宣言したかどうかなど覚えていられません[i]。その時に for 文の中でループ用の変数を宣言するという荒技が可能です。これは変数宣言の位置が C 言語だとブロックの最初と規定されていたのが、C++だと変数宣言の位置は自由[ii]となったためできるようになったためです。変数が使える範囲は変数宣言をした場所からその宣言したブロックの最後までです。もちろん C 言語のように関数の最初に必要な変数を宣言するのが基本ですが、関数の中でもごく一部でしか使わない変数に関してはその場で宣言するというのはアリです。

次のようなコードを見てみましょう。

```
1   #include<iostream>
2   using namespace std;
3   int main(void){
4       for( int i = 0; i < 10; i++){
5           for( int i = 0; i < 10; i++){
6               cout << "i = " << i << endl;
7           }
8       }
9       return 0;
10  }
```

for 文で i を宣言し、その中の for 文で変数 i を宣言しています。この場合、入れ子の中の cout の i はどちらの i でしょうか？　実行して試してみてください。

第三節　関数のオーバーロード

次に重要なものとして関数のオーバーロードという機能がつきました。これは同じ関数名であっても引数の型が違えば同名の関数を複数個作ることができるという機能です。

```
1   #include<iostream>
2   using namespace std;
3
4   int add( int x, int y){
5       return x + y;
6   }
7   double add( double x, double y){
8       return x + y;
9   }
10
11  int main(void){
12      int a, b;
13      double c, d;
14      cin >> a;
15      cin >> b;
16      cin >> c;
17      cin >> d;
```

---

i　関数は 100 行に収まる程度というのが目安なので、宣言した変数くらい把握しておかなければならないが……

ii　VC だと微妙に挙動がおかしいんだけどね……

```
18      cout << add( a, b) << endl;
19      cout << add( c, d) << endl;
20      return 0;
21  }
```

cin[i] がちょっと多くてアレですが、add という関数を 2 つ宣言していることがわかると思います。一つは引数の型がどちらも int 型、もう片方は引数の型が double 型となっています。このような仕組みを関数のオーバーロードと言います。このコードは C 言語のコンパイラだとエラーがでますが、C++のコンパイラならエラーがでません。このオーバーロードの利点は引数の型を変わったとしても似たような意味合いを持つ関数ならば同じ名前で統一できるという利点があります[ii]。但しこの関数のオーバーロードはコンパイラによって実現しているので動的な変更[iii]ができないという欠点があります。

第四節　その他 C++で追加された機能

他に C++では予約語が増えました。主な予約語[iv]は「new」、「delete」、「bool」などです。詳しくはその手の本で調べてください。また新しい型として bool 型[v]というのが増えました。真か偽かだけの型で主に if 文で使えます。

ポインタに代わり参照型というのも増えました。これは従来のポインタが C 言語の元来の使用方法 OS を記述するためであったため、ポインタがあまりにも強力すぎたため下手に使うと OS ごとクラッシュさせる要因になっていました。そのため、関数の間で変数をやりとりする際などにポインタを用いずに安全に受け渡しができるようにと参照型というのが作られました。この参照型というのは最近のほとんどの言語で実装しており、その代わりポインタを実装しないという言語が増えているようです。ただ参照型はポインタの劣化コピーみたいなもの[vi]なので、今回はポインタのみを使ってコードを書きます。

このテキストでは触れませんがテンプレートという強力な機能が追加されています。これは引数の型に依存せずに関数を記述できる機能です。プリプロセッサマクロと似たような機能ですが、テンプレートのほうが関数と扱え、また必定以上のコードを消費しないという利点があります。

またテンプレートを利用したライブラリとして STL(Standard Template Library)という強力なライブラリが存在します。この STL は動的配列、リスト、連想配列などなど様々な機能がライブラリとして提供されています。この STL はテンプレートの機能を最大限に生かされて作られた非常に強力なライブラリで、使いこなすようになると同じ C++のコードでもまるで違う世界を作り出してくれる非常に面白いライブラリです。

その他いろいろ増えましたが、基本的な機能は全て C 言語に準じているので必要な機能だけ C++を使い、普段は C で書くといった方法でも十分な C++のプログラミング手法に一つです[vii]。

---

i　ちなみに cin の場合は引数で参照型を使っているため変数の前に&は要らない

ii　これくらいの規模だと関数マクロを使うし、これくらいの差異ならテンプレートを使うことになるが……

iii　実行中に使う関数を判断するということ。関数ポインタなどを使えば実現できる

iv　C 言語なら「for」とか「if」とか。VC で入力して色が変わった言葉が予約語という解釈で大丈夫

v　int でいいじゃんって思うのは筆者だけだろうか

vi　twitter でそう呟いたら、色々詳しい人が教えてくれた。ググるよりも簡単に情報が手に入れられる twitter

vii　betterC と呼ばれる。ちなみに C++の仕様を理解しているプログラマは一握りだとか

第三章　クラス

前置きが長くなりましたが、実際にクラスの作成に移っていきます。オブジェクト指向の用語を容赦なく使っていくので非常に混乱するとは思いますが雰囲気で突き進んでください。

第一節　構造体

C++のクラスは構造体を拡張する形で実現しています。なのでまずは構造体のおさらいをします。構造体の作りかたは下記の通りです。

```
struct name{
    int variant1;
    int variant2;
    //...以下略
};
```

構造体は新しく型を作っているのと同じことです。int 型や double 型は C 言語、及び C++に予めある型なので組み込み型と呼ばれます。一方、構造体で作った型はユーザ定義型と呼ばれます。ユーザ定義型はプログラマが作った型の他にも ANSI C で定義されている型も含まれます[i]。上記の例では name という型を作り、その型の中身は varinat1 と variant2 と呼ばれる変数で構成されています。

実際にコードに書いてみましょう。

```
1   #include<iostream>
2   using namespace std;
3   struct point{
4       int x;
5       int y;
6   };
7
8   int main(void){
9       point a, b;
10      a.x = 10; a.y = 20;
11      b = a;
12      cout << "b.x = " << b.x << endl;
13      cout << "b.y = " << b.y << endl;
14      return 0;
15  }
```

先ほどの通り構造体はユーザ定義型ですが、実際は普通の型と何ら変わりなく使用することができます。C 言語と違い C++は構造体を使用した変数の宣言時に strcut は省略可能となりました[ii]。よって「point a, b;」という point 型の a と b という変数を作るという意味となります。この辺は「int a, b」と宣言した場合は int 型の a と b という変数を作るのと変わりません。また変数の型の代入ができるので「b = a」といった一括代入[iii]ができます。ですが残念なことに足し算や引き算などの演算についてはできません。できる演算機能は代入だけです[iv]。

次に構造体のポインタについてです。

---

i　ようは構造体で初めから作ってある型がある。日付型とか

ii　C 言語では「typedef」をつけなければならない

iii　配列は一括代入ができない

iv　先ほどもチラッとでたが演算子のオーバーロードたるもので解決できる

```
1   #include<iostream>
2   using namespace std;
3   struct point{
4       int x;
5       int y;
6   };
7
8   int disp( point* input){
9       cout << input->x << endl;
10      cout << input->y << endl;
11      return 0;
12  }
13  int main(void){
14      point a;
15      a.x = 10; a.y = 20;
16      disp( &a);
17      return 0;
18  }
```

いい感じに難しくなってきました。特に disp 関数あたり。disp 関数の引数は point 型のポインタを引数にとっています。ポインタとして受け取った構造体のメンバ変数を cout を用いて表示していることになります。具体的には main 関数で宣言した変数 a のアドレスを受け取ってそのポインタを使い変数 a にアクセスしていることになります。

ポインタと聞くとアスタリスク「*」を思い浮かべます。確かに構造体ではないポインタに関してはアスタリスクを用いるのが正解なのですが、この場合「*input.x」と書いても input メンバのポインタ変数 x という扱いになってしまって input ポインタが指すメンバ x にアクセスすることができません。アスタリスクを用いる場合は括弧を使用しなければならない[i] ため通常、構造体のポインタはアロー演算子「->」と呼ばれるものを使います。これはクラスになると山のように使うので覚えておいてください[ii]。

第二節　構造体からクラスへ

構造体をクラスに変えていきます。方法は簡単です。

```
1   class point{
2       int x;
3       int y;
4   };
5
6   int main(void){
7       point a, b;
8       //後は面倒なので略
9       return 0;
10  }
```

struct を class と置き換えただけです。これでめでたくコンパイルはできるのですが、これだと「a.x」にアクセス[iii]しようとするとエラーがでます。これは struct のデフォルトアクセス指定

---

i　具体的には(*input).x と書かなければならないため非常にややこしい

ii　つまりポインタを山のように使うということ

iii　その値に値を取得したり代入しようとしたりすること

子が public に対し、class のデフォルトのアクセス指定子が private になっている[i]から起こることです。次のように一文を追加してください。

```
class point{
public:         //←この一行を追加
    int x;
    int y;
};
```

変数を int で宣言する前に「public:」という一文を付け足すことによって、それ以降の変数やメソッドのアクセス制限が public となります。

```
class TestClass{
public:                     //ここからpublic
    int var1;
    double var2;
    int method1(void);      //ここまでがpublic
private:                    //ここからprivate
    int var2;
    int method2(void);      //ここまでprivate
};
```

private と public の違いは自分のクラスからアクセスできるか、できないかの違いです[ii]。基本的に他から不用意に変数を変えられてしまうのを防ぐために private で極力宣言しなければなりません[iii]。これをカプセル化と言ってオブジェクト指向の大事な要素の一つとなっています。自クラスの中の変数は自クラスのメソッドからアクセスし、他のクラスや関数との混同を避ける必要があります。何故かというと他の関数やクラスに依存してしまうと他のアプリケーションでそのクラスを流用しようとしたときにせっかく作ったクラスが無意味になってしまう可能性が高いからです。

しかし、一つデメリットができます。他のクラスや関数からアクセスされる必要がある変数はどうすればいいのかという問題です。解決法の一つに先のように極力 private で宣言し、他からアクセスされる変数だけ public で宣言するというものです。これは単純ではありますが、例えば他クラスからの変数の読み込みは OK だけど、書き込みはダメといったことには対応できません。

もう一つの解決方法にアクセサ、いわゆるゲッタ、セッタを作るというものがあります。変数を取得するメソッドを public で宣言するという方法です。get ～、set ～というメソッド名をつけることが習慣的なことからゲッタ、セッタと呼ばれます[iv]。このアクセサは非常に便利で内部での仕様変更をクラス内だけに抑える[v]という役目を持っています。ただこれも問題点があり、C++では各変数に対応したメソッドをその都度書かなければならないという点です。中身は 3 行で済むというのはわかると思いますが、面倒極まりないです。

余談ですが某窓も面倒と思ったのか C#ではこのゲッタ、セッタを言語として自動生成する仕

---

i　struct のデフォルトのアクセス指定子は public となっている

ii　デフォルトのアクセス指定子が public のため、何も書かなくても構造体ではエラーがでなかった

iii　個人的には private よりも protected をお奨めする。private はアクセス範囲が狭すぎる

iv　WindowsAPI を見ると get ～、set ～という関数を大量に目にすることができる

v　単に情報取得の関数もゲッタセッタで作る。例えば今までクラス内の座標管理を直行座標系だったのを極座標系に変えたとする。また XY の座標を取得するメソッド、getCoordinate メソッドがあったとして、クラス内は全て極座標系に変更したとしても getCoordinate メソッドの内部で極座標系から直交座標系に変換してやれば外部の関数などは変更しなくてもよくなる

様となっています。

制御子	アクセス範囲
public	全ての場所からアクセスできる
private	クラスのメンバ関数とフレンド関数からしかアクセスできない
protected	private のアクセス範囲と、public の派生クラスからアクセスすることができる。

今回のテキストでは以上の通り、カプセル化はオブジェクト指向の概念で非常に重要なことですが、煩雑化を防ぐために全ての変数、メソッドを public で書くこととします。

第三節　メソッド

次にオブジェクト指向というのはクラスにメソッドを追加しなければなりません。なのでメソッドを追加します。通常はヘッダと実行部に分ける必要がありますが今回はヘッダ部にそのままメソッドを書きました。分ける書き方は後のほうで紹介します。

```
1   #include<iostream>
2   using namespace std;
3   class point{
4   public:
5       int x;
6       int y;
7       int disp(void){
8           cout << this->x << endl;
9           cout << this->y << endl;
10          return 0;
11      }
12  };
13
14  int main(void){
15      point a;
16      a.x = 10; a.y = 20;
17      a.disp();
18      return 0;
19  }
```

disp メソッドが追加されています。これは先の disp 関数と同じ役割を持っています。ですが構造体のポインタを引数に取るのではなく、this ポインタというものを使っています。this ポインタというのは自分自身のインスタンスのことを指し、この場合は自分自身のインスタンス内の変数 x を参照しろという意味です。おそらく今までの関数と書き方はほぼ変わりないので難しいところは特にないと思います。

ここで少し小難しい話[i]をします。上記のプログラムのように「point　a」と宣言されるとスタックという領域に変数は作られます。これは上限が決められており VisualC++の場合だと 1MByte になっています。多いか少ないかでいうと現在の PC のメインメモリの容量が GByte 単位になっていることを考えると非常に少ないと思われます。スタック領域以外にヒープ領域というのが存在します。これは PC のメインメモリの容量ぎりぎりまで使うことができます[ii]。しかしこのヒ

i　今までの話は小難しくない

ii　ギリギリまで使うと Windows に怒られるんだけどね

ープ領域を使うには通常の変数の宣言のやり方ではなく new 演算子や malloc 関数を使わなければなりません。詳しくはエレ研の 2007 年機関誌を読んでください。

インスタンスは通常ヒープ領域に作ります。なので上記のプログラムは間違いではないですが、C++っぽくないプログラムです。なのでヒープ領域にインスタンスを作るプログラムを紹介します。

```
1   #include<iostream>
2   using namespace std;
3   class point{
4   public:
5       int x;
6       int y;
7       int disp(void){
8           cout << this->x << endl;
9           cout << this->y << endl;
10          return 0;
11      }
12  };
13
14  int main(void){
15      point* a;
16      a = new point;
17      a->x = 10; a->y = 20;
18      a->disp();
19      delete a;
20      return 0;
21  }
```

オブジェクトの変数をポインタ型でおき、new 演算子を使ってインスタンスを作成しています。このときポインタ型なのでインスタンス変数やメソッドへはアロー演算子を使ってアクセスします。new 演算子、delete 演算子というのはメモリ領域を確保、解放[i]する演算子で、C 言語の malloc 関数と free 関数のようなものです。malloc 関数、free 関数との違いは new、delete はコンストラクタ、デストラクタ(後述)が自動で呼び出されるのに対し、malloc 関数と free 関数は明示的にコンストラクタ、デストラクタを呼び出さなければならない点です。特別なこと[ii]がない限り C++では new 演算子、delete 演算子を用います。

第四節　コンストラクタ、デストラクタ

コンストラクタ、デストラクタというのはインスタンス生成時と破棄時に自動的に呼び出される特別なメソッドのことを指します。

```
1   #include<iostream>
2   class hoge{
3       hoge( void){
4           cout << "constructer" << endl;
5       }
6       ~hoge( void){
7           cout << "destructer" << endl;
8       }
```

i　解放しない場合はずっとその領域を使い続ける、メモリリークという現象が起きる。最悪の場合 OS ごと落ちる

ii　特別なことの例：「n」「e」「w」いずれかのキーが壊れた等

```
9   };
10
11  int main( void){
12      hoge* me;
13      me = new hoge();
14      cout << "newしました" << endl;
15      delete me;
16      cout << "deleteしました" << endl;
17      return 0;
18  }
```

コンストラクタはクラス名と同名、デストラクタはクラス名の頭にチルダ「~」を付けたメソッド名にしなければなりません。上記のクラスはコンストラクタ呼び出し時に「constructer」と表示し、デストラクタ呼び出し時に「destructer」と表示します。コンストラクタ、デストラクタの特徴として戻り値が一切ないという点です。void 型でもないので return を書くこともできません。コンストラクタは主にインスタンス変数の初期化に使われ、デストラクタは主にインスタンスが持っているクラスの破棄時に使われます。

コンストラクタは引数なしのものをデフォルトコンストラクタといいますが、引数があるものを作ることも可能です。また引数がないものとあるものの二つを記述することも可能です。

```
class hoge{
    hoge( void){
        cout << "constructer 1" << endl;
    }
    hoge( int n){
        cout << "constructer 2" << endl;
    }
    //以下略
};
```

一章二節で紹介した C++の新しい機能、オーバーロードによって実現しています。インスタンス生成時の new するときに引数がないと「constructer 1」と表示され、引数があると「constructer 2」と表示されます。引数を使ってインスタンス変数の初期値をあらかじめ決めることができます。もちろん三つ以上のコンストラクタも作ることができます。オーバーロードは全ての関数、メソッドに適用できるため、デストラクタやインスタンスメソッド、クラスメソッドもオーバーロードすることができます。

第五節　少し実践

ここで少し実践的なプログラミングをしたいと思います。

例題として分数の計算を行うプログラムの製作を行いたいと思います。実装する機能は下記の通りです。

- **分数の入力及び出力**
- **分数同士の加算、減算、乗算、除算**
- **分数の約分**

一応それなりの量なので読むのは結構つらいと思います。

```
1   //main.cpp
2
3   #include"main.h"
4
5   int main(void){
```

```
6       fraction* fract[3] = { 0, new fraction(), new fraction()};
7
8       fraction* (*calc[])( const fraction*, const fraction*) = { fraction::add, frac
        tion::sub, fraction::mult, fraction::div};
9       //ついカッとなって関数ポインタを使った
10
11      for( int i = 1; i < 3; i++){
12          fract[i]->set_num_den();
13          fract[i]->disp();
14      }
15
16      {
17          int num;
18          printf("1.足し算 2.引き算 3.かけ算 4.割り算¥n");
19          scanf( "%d", &num);
20          fract[0] = calc[num - 1](fract[1], fract[2]);
21          fract[0]->disp();
22      }
23
24      delete fract[0];
25      delete fract[1];
26      delete fract[2];
27      return 0;
28  }
29
```

```
1   //main.h
2
3   #include<stdio.h>
4
5   #ifndef ___main
6   #define ___main
7
8   #include"fraction.h"
9
10  #endif
11
```

```
1   //fraction.cpp
2
3   #include"fraction.h"
4
5   fraction::fraction(void){
6       num = 0;
7       den = 1;
8   }
9
10  fraction::~fraction(void){
11
12  }
13
14  int fraction::disp(void){
15      printf("%d / %d¥n", num, den);
16      return 0;
17  }
```

```
18
19  int fraction::set_num_den(void){
20      printf("数字を入力してください¥n");
21      scanf("%d %d", &num, &den);
22      return 0;
23  }
24
25  int fraction::reduction(void){
26      for(int i = num; i > 0; i--){
27          if( den % i == 0 && num % i == 0){
28              num /= i;
29              den /= i;
30          }
31      }
32      return 0;
33  }
34
35  int fraction::lst_com_mult(int x, int y){
36      //LeastCommonMultiple、最小公倍数を求める
37      for( int i = x; i > 0; i--){
38          if( x % i == 0 && y % i == 0){
39              //最大公約数を求める
40              return (x / i) * (y / i) * i;
41          }
42      }
43      return 1;
44  }
45
46  fraction* fraction::add(const fraction* input1, const fraction* input2){
47      fraction *output = new fraction();
48      fraction *buf = new fraction();
49      int den = lst_com_mult( input1->den, input2->den);
50
51      *buf = *input1;
52      buf->num = buf->num * den / input1->den;
53      buf->den = buf->den * den / input1->den;
54
55      *output = *buf;
56
57      output->num += input2->num * den / input2->den;
58
59      delete buf;
60      output->reduction();
61      return output;
62  }
63
64  fraction* fraction::sub(const fraction* input1, const fraction* input2){
65      fraction *buf;
66      *buf = *input2;
67      buf->num *= -1;
68      return fraction::add( input1, buf);
69  }
70
71  fraction* fraction::mult(const fraction* input1, const fraction* input2){
72      fraction *output = new fraction();
```

```
73      output->num = input1->num * input2->num;
74      output->den = input1->den * input2->den;
75      output->reduction();
76      return output;
77  }
78
79  fraction* fraction::div(const fraction* input1, const fraction* input2){
80      fraction *output = new fraction();
81      output->num = input1->num * input2->den;
82      output->den = input1->den * input2->num;
83      output->reduction();
84      return output;
85  }
86
```

```
1   //fraction.h
2
3   #include"main.h"
4
5   #ifndef ___fraction
6   #define ___fraction
7
8   class fraction{
9   public:
10      fraction(void);
11      fraction( int num, int den);
12      ~fraction(void);
13
14      int num;
15      //分子
16
17      int den;
18      //分母
19
20      int disp(void);
21      //現在の分数の値を表示する
22
23      int set_num_den(void);
24      //値を決める
25
26      int reduction(void);
27      //約分する
28
29      static int lst_com_mult( int x, int y);
30      //LeastCommonMultiple、最小公倍数を求める
31
32      static fraction* add( const fraction*, const fraction*);
33      //足し算
34
35      static fraction* sub(const fraction*, const fraction*);
36      //引き算
37
38      static fraction* mult(const fraction*, const fraction*);
39      //かけ算
40
```

```
41      static fraction* div(const fraction*, const fraction*);
42      //割り算
43  };
44
45  #endif
46
```

ここまで。

プログラムの解説ですがわりと fraction.cpp と fraction.h が分数のクラスとなっています。.cpp のほうを実装部、.h のほうをインターフェイス部といいます。通常はこのように一つのクラスは 2 つのファイルからなっていることが非常に多いです。インターフェイス部には構造体を作ったときと同じように変数を宣言したり、メソッドを宣言する場所です。ここではメソッドのプロトタイプ宣言だけなので実際の実装は行いません。ただインターフェイス部にメソッドの実装を行うこともできるので、このような紙面の節約のため[i] よくそのまま実装されます。.cpp のほうは実装部です。ようはメソッドの中身を記述する場所です。メソッドの実装の仕方は

戻り値 クラス名::メソッド名(引数の型 変数名, 引数の型 変数名, ...){...}

となります。関数と違うのはメソッドが属するクラスをメソッド名の前に書き、コロンを後ろに 2 つつける[ii]ところです。

またいくつかのメソッドのインターフェイス部には static と書いてあるのがわかります。これは静的メソッド、つまりクラスメソッドを表します。第一章でも説明しましたがクラスメソッドとはインスタンスに関係ないクラスのメソッドです。イメージとしてはクラスの中の普通の関数と言ったところでしょうか。今回は計算関係の処理を全てクラスメソッドとして記述し、二つの fraction クラスを引数にとり、新たなインスタンスを戻り値としています。

実際の具体的な中身についてですが今回のインスタンス変数は分子と分母を表す、int num と int den だけです。それぞれ分子と分母の値に対応します。インスタンスを生成するときにコンストラクタで num = 0、den = 1 つまり 0/1 で初期化します。次に生成したインスタンスに値を代入します。これはインスタンスメソッドである set_num_den メソッドが担当します。main.cpp 側から値を代入したいインスタンスの set_num_den メソッドを呼び出すことでそのインスタンスに値を代入することができます。計算を行うときはクラスメソッドである add メソッドなどを用います。これらは全てポインタ引数をとるので const をつけて値の破壊を防いでいます。戻り値は add メソッドなど内で生成したインスタンスのアドレスが戻り値となります。よって main.cpp 内ではそのアドレスを受け取れるように fraction ポインタで戻り値を受け取ります。

分数の計算の処理についての説明は割愛します。おそらく小学 4 年生レベルの知識があれば何をやっているかはわかるはずです[iii]。

第六節　継承

すでに作ってあるクラスを拡張、修正などをするためには継承というオブジェクト指向の機能を使います。

```
1   #include<iostream>
2   class article{
```

---

i　インターフェイス部と実装部分けるだけで 5 行くらい増えちゃうからね

ii　スコープ解決演算子ってやつ

iii　単に最小公倍数とかの説明が面倒だっただけだけど

```
3   public:
4       int x;
5       int y;
6
7       article(void){
8           x = 0;
9           y = 0;
10          cout << "article Constructor" << endl;
11      }
12      ~article(void){
13          "article Destructor" << endl;
14      }
15      int disp(void){
16          cout << x << endl;
17          cout << y << endl;
18      }
19  };
20
21  class move : public article{
22  public:
23      double velocity;
24      double angle;
25
26      move(void){
27          velocity = 0.0;
28          angle = 0.0;
29          cout << "move Constructor" << endl;
30      }
31      ~move(void){
32          cout << "Destructor"" << endl;
33      }
34      void update{
35          x += velocity * cos(angle);
36          y += velocity * sin(angle);
37      }
38  };
39
40  int main(void){
41      move* bullet;
42      bullet = new move();
43      bullet->velocity = 1.0;
44      do{
45          bullet->angle += 0.1;
46          bullet->update();
47          bullet->disp;
48      }while(1);
49      delete bullet;
50      return 0;
51  }
```

article というクラスは X 座標、Y 座標しか保持できないクラスです。article クラスを継承して、速度と向きを保持する変数や関数を追加した move というクラスを作りました。これはすでに X 座標、Y 座標という変数は article クラスで宣言されているので宣言する必要はありません。article クラスの全ての変数、メソッドを引き継いでそれに足す形で速度と向きを保持するクラスを作っ

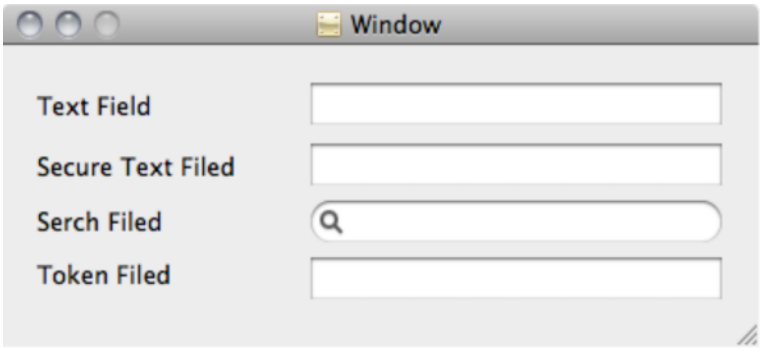
ています。

またコンストラクタ、デストラクタの挙動についても注目してください。派生クラスのインスタンスを生成した時点でコンストラクタが呼び出されます。順序は基底クラス→派生クラスの順番です。基底クラスのコンストラクタの先に呼び出さないと派生クラスで基底クラスのインスタンス変数を使う可能性があるからです。逆にデストラクタの順は派生クラス→基底クラスの順番となります。

継承を利用することでのメリットは次の二つです。一つめに重複するような機能、メソッドを一カ所に管理することによって仕様の変更や周りの関数との連携をとりやすくすることができる。二つめに一度作ったクラスを再利用しやすくなり、すでにあるメソッドを一部特殊な用途に使いたいときにそのメソッドだけを修正することができます。

まず一つめの重複する機能を一カ所に管理するというのを考えてみましょう。まず基本となるスーパークラスに基本的な機能を全ていれます。例えば指定した座標に画像を表示するメソッドや保有している画像を管理するメソッドです。そこから継承して速度や向きの変数、またそれらを制御するメソッドを追加することができます。座標の管理を絶対座標、相対座標といった違いは全てスーパークラスに任せ、サブクラスは絶対座標のみで記述といったことができるようになります。

二つめの一度作ったクラスの再利用を見てみます。継承を用いることによってすでに作ってあるクラスをもう一度使い直すことが容易となります。例えばテキストボックスをみると、通常のテキストボックスはキーボードから入力した文字がそのまま画面へと表示されます。しかしパスワードを入力する場面では黒丸へと自動置換される機能を持っています。これはまずテキストボックスのクラスを作り、そのあとにパスワード用のテキストボックスを作って入力した文字をそのまま表示させないようなメソッドに修正することで可能となります。



第七節　仮想関数

また基底クラスのメソッドと同名のメソッドを派生クラスで作ることができます。このことをオーバーライドといいます。すでにあるメソッドを書き換えることができるので、あるクラスの変更を行うときにオーバーライドを使います。

予めオーバーライドを前提としたメソッドに virtual とつけることによってその関数をオーバーライドすることができます。その virtual とつけたメソッドのことを仮想関数と呼びます。しかし実際は virtual とつけなくてもコンパイラが仮想関数と勝手に見なしてくれるので、virtual と明示しなくてもオーバーライドすることができます。

```
1  #include<iostream>
2  using namespace std;
3  class base{
```

```
4   public:
5       int method(void){
6           cout << "Base Class" << endl;
7           return 0;
8       }
9   };
10  class inheritance: public base{
11  public:
12      int method(void){
13          cout << "Inherited Class" << endl;
14          return 0;
15      }
16  }
17
18  int main(void){
19      inheritance* instance = new inheritance();
20
21      instance->method();             //派生クラスのmethodを呼び出す
22      instance->base::method();       //基底クラスのmethodを呼び出す
23
24      delete instance;
25      return 0;
26  }
```

実行結果は「Base Class　Inherited Class」となります。21 行目でオーバーライドした method を呼び出し、そこでは派生クラスの method が呼び出されます。次に 22 行目ではスコープ解決演算子を用いて基底クラスの method を呼び出しています。同じメソッド名のため「(クラス名)::(メソッド)」という文法になります。またメソッドを上書きではなく拡張という概念とすると次のような使い方もできます。

```
1   #include<iostream>
2   using namespace std;
3   class base{
4   public:
5       int method(void){
6           cout << "Base Class" << endl;
7           return 0;
8       }
9   };
10  class inheritance: public base{
11  public:
12      int method(void){
13          cout << "Inherited Class" << endl;
14          return base::method;
15      }
16  }
17
18  int main(void){
19      inheritance* instance = new inheritance();
20
21      instance->method();             //派生クラスのmethodを呼び出す
22
23      delete instance;
24      return 0;
25  }
```

14 行目で return するといに基底クラスを呼び出しています。これによって今までのメソッドを拡張する形でオーバーライドをしています。14 行目で基底クラスのメソッドを呼び出していますが、メソッドの中身によってはメソッドの最初の行に基底クラスを呼び出すのもありです。

参考文献

プログラミング講義 C++　柴田望洋　ソフトバンクパブリッシング
たのしい Cocoa プログラミング　木下誠　BugNewsNetwork
詳解 Objective-C2.0　荻原剛志　ソフトバンクパブリッシング
ゲームプログラマになる前に覚えておきたい技術　平山尚　秀和システム
2008 年度のガス先輩のテキスト