C++によるオブジェクト指向

Object Oriented software construction in C++

－　２００２年度版　－

福岡工業大学
情報工学部 情報工学科
柴田望洋

BohYoh Shibata
Fukuoka Institute of Technology

本資料について

◆　本資料は、2002 年度・福岡工業大学 情報工学部 情報工学科４年生の講義

『ソフトウェア工学』

の補助テキストとして、福岡工業大学 情報工学部 情報工学科 柴田望洋が編んだものである。

◆　参考文献・引用文献等は、資料の最後にまとめて示す。

◆　諸君が本資料をファイルに綴じやすいように、研究室の学生達（卒研生と大学院生）が時間を割いて、わざわざ穴を開けるという作業を行っている（一度のパンチで開けることのできる枚数は限られており、**気の遠くなるような**時間がかかっている）。
必ずＢ５のバインダーを用意して、きちんと綴じていただきたい。

本資料は、Microsoft 社のワープロソフトウェアである Microsoft Word 2002 を用いて作成した。

★　本講義では、以下に示すホームページ上の、各ドキュメントも参照するので、参考にされたい。

柴田望洋後援会オフィシャルホームページ　http://www.BohYoh.com/

じゃんけんゲーム

まずは、じゃんけんゲームを作りましょう。最初は単純なものを作り、少しずつ複雑にしていきます。

■ じゃんけんゲーム（Ver.1）

```
/*
    じゃんけんゲーム（Ver.1）
*/

#include  <ctime>
#include  <cstdlib>
#include  <iostream>

using namespace  std;

int main(void)
{
    int  user;            // ユーザの手
    int  comp;            // コンピュータの手
    int  retry;           // もう一度？
    time_t t;             // 時刻

    srand(time(&t) % RAND_MAX);     // 乱数の種を初期化

    cout << "じゃんけんゲーム開始。¥n";

    do {
        comp = rand() % 3;    // コンピュータの手（0〜2）を乱数で生成

        cout << "¥n¥aじゃんけんポン … (0)グー (1)チョキ (2)パー：";
        cin >> user;

        cout << "私は";       // コンピュータの手を表示
        switch (comp) {
         case 0: cout << "グー";     break;
         case 1: cout << "チョキ";       break;
         case 2: cout << "パー";     break;
        }
        cout << "です。¥n";

        int diff = (user - comp + 3) % 3;

        if (diff == 0)
            cout << "引き分けです。¥n";
```

```
        else if (diff == 1)
            cout << "あなたの負けです。¥n";
        else
            cout << "あなたの勝ちです。¥n";

        cout << "もう一度しますか … (0)いいえ (1)はい：";
        cin >> retry;
    } while (retry == 1);

    return (0);
}
```

プログラムは、最近 ANSI / ISO で規格制定された C++にのっとっています。したがって、

講義『プログラミング言語』

で学習したものと、少し言語仕様などが異なります。

ヘッダ名

C++のヘッダは<iostream.h>でなく<iostream>となり.h がなくなります。

C 言語のヘッダは、<time.h>が<ctime>となり、先頭に c が付き、.h がなくなります。

名前空間

cout に対する出力や、cin からの入力などの標準的なライブラリを利用するには、

```
using namespace std;
```

が必要です。

このプログラムでは、グー、チョキ、パーの各“手”を、それぞれ 0, 1, 2 の整数で表していることは分かりますね。

変数 comp がコンピュータの手です。乱数によって生成します。乱数の発生については、情報工学ゼミナールで学習しましたね。

変数 user はユーザの手です。キーボードから入力します。

勝ち負けの判定は、割り算をうまく使って行います。

“手”の実現

さて、ユーザが手を入力した後に、

私はグーです。

と表示しますが、これを、

私はグーで、あなたはパーです。

と表示するように改良してみましょう。プログラムは、以下のようになります。

■ じゃんけんゲーム（Ver.1.1 部分）

```
// 省略
        cout << "私は";       // コンピュータの手を表示
        switch (comp) {
         case 0: cout << "グー";     break;
         case 1: cout << "チョキ";       break;
         case 2: cout << "パー";     break;
        }
        cout << "で、あなたは";
        switch (user) {
         case 0: cout << "グー";     break;
         case 1: cout << "チョキ";       break;
         case 2: cout << "パー";     break;
        }
        cout << "です。¥n";

// 省略
```

このプログラムでは、手を表す文字列〝グー〟、〝チョキ〟、〝パー〟が三つずつ現れます。もちろん、今後の仕様の変更などに伴って、さらに増える可能性もあるかもしれません。

したがって、これらの文字列は文字列の配列に格納することにした方がよいでしょう。文字列の配列は、次のように宣言できます。

```
char  hd[][7] = {〝グー〟, 〝チョキ〟, 〝パー〟};
```

このようにすると、hd[0]は〝グー〟、hd[1]は〝チョキ〟、 hd[2]は〝パー〟となり、添字と手を表す整数とが一致するため、都合がよいですね。

※　環境によっては、日本語の文字列を表せない場合や、たとえば〝チョキ〟の占有領域が 7 バイトでは不足する場合もあります。

手を表す文字列の配列を導入して改良・変更したプログラムを次ページに示します。

■ じゃんけんゲーム（Ver.1.2）

```
/*
    じゃんけんゲーム（Ver.1.2：手の文字列を導入）
*/

#include  <ctime>
#include  <cstdlib>
#include  <iostream>

using namespace  std;

int main(void)
{
    int  user;              // ユーザの手
    int  comp;              // コンピュータの手
    int  retry;             // もう一度？
    char hd[][7] = {"グー", "チョキ", "パー"};
    time_t t;               // 時刻

    srand(time(&t) % RAND_MAX);      // 乱数の種を初期化

    cout << "じゃんけんゲーム開始。¥n";

    do {
        comp = rand() % 3;    // コンピュータの手（0～2）を乱数で生成

        cout << "¥n¥aじゃんけんポン … ";
        for (int i = 0; i < 3; i++)
            cout << "(" << i << ")" << hd[i] << " ";
        cout << "：";

        cin >> user;

        cout << "私は"      << hd[comp] << "で、"
             << "あなたは" << hd[user] << "です。¥n";

        int diff = (user - comp + 3) % 3;

        if (diff == 0)
            cout << "引き分けです。¥n";
        else if (diff == 1)
            cout << "あなたの負けです。¥n";
        else
            cout << "あなたの勝ちです。¥n";

        cout << "もう一度しますか … (0)いいえ (1)はい：";
        cin >> retry;
    } while (retry == 1);

    return (0);
}
```

改良に伴い、
むしろ複雑になった

改良に伴い、
非常に簡潔になった

関数の分割

ここまでのプログラムは、すべてを main 関数だけで賄なおうとするものでした。このようなプログラミングでよいのでしょうか。

将来的に、たとえば、

■　勝敗の回数表示

■　三人のじゃんけんにする（コンピュータが２人）

といったバージョンアップを行うとするとどうでしょう。プログラムは大きくなりますので、すぐに破綻をきたすことは明らかです。

また、“手”を表す文字列の配列 hd の導入に伴って、

じゃんけんポン　…(0)グー (1)チョキ (2)パー

と表示する部分も非常に見づらいものとなっています。

したがって、機能別に関数としてまとめるべきであることが分かります。

まずは、無理矢理（？）関数に分割したプログラムを示します。

■ じゃんけんゲーム（Ver.2.0）

```
/*
    じゃんけんゲーム（Ver.2.0：関数の分割）
*/

#include  <ctime>
#include  <cstdlib>
#include  <iostream>

using namespace  std;

int  user;              // ユーザの手
int  comp;              // コンピュータの手
char hd[][7] = {"グー", "チョキ", "パー"};

/*--- じゃんけん実行 ---*/
void jyanken(void)
{
    comp = rand() % 3;    // コンピュータの手（0～2）を乱数で生成

    cout << "¥n¥aじゃんけんポン … ";
    for (int i = 0; i < 3; i++)
        cout << "(" << i << ")" << hd[i] << " ";
    cout << "：";

    cin >> user;
}
```

宣言が関数の外に移された

```
/*--- 手を表示 ---*/
void disp_hands(void)
{
    cout << "私は"     << hd[comp] << "で、"
         << "あなたは" << hd[user] << "です。¥n";
}

/*--- 判定結果を表示 ---*/
void disp_result(int result)
{
    if (result == 0)
        cout << "引き分けです。¥n";
    else if (result == 1)
        cout << "あなたの負けです。¥n";
    else
        cout << "あなたの勝ちです。¥n";
}

/*--- 再挑戦するかを確認 ---*/
int confirm_retry(void)
{
    int  x;

    cout << "もう一度しますか … (0)いいえ (1)はい：";
    cin >> x;

    return (x);
}

int main(void)
{
    int  retry;                  // もう一度？
    time_t t;                    // 時刻

    srand(time(&t) % RAND_MAX);     // 乱数の種を初期化

    cout << "じゃんけんゲーム開始。¥n";

    do {
        jyanken();               // じゃんけん実行

        disp_hands();            // コンピュータとユーザの手を表示

        int result = (user - comp + 3) % 3;

        disp_result(result);     // 判定結果を表示

        retry = confirm_retry(); // 再挑戦するかを確認

    } while (retry == 1);

    return (0);
}
```

■　関数の分割によって、main 関数がすっきりしましたね。以下にも示すように、注釈を読むだけで、プログラムのおよその流れが分かります。

```
do {
    // じゃんけん実行
    // コンピュータとユーザの手を表示
    // 判定
    // 判定結果を表示
    // 再挑戦するかを確認
} while (retry == 1);
```

なお、ほとんどの変数の宣言が、main 関数の外に移されたことに注意しましょう。
※このようなやり方がよいかどうかは別ですが。

■　じゃんけんの勝敗結果を格納する変数名を diff から result に変更しました。もともと diff は、difference の略であり、ユーザの手からコンピュータの手の値を**引く**ことから命名したものです。
しかし、このような名前は、適当なものであるとはいえません。というのも、その変数の意味を表していないからです。

■　main 関数の先頭では、乱数の種を初期化していますが、この箇所だけが、直接プログラムがかかれており、その点では具体的ですね。ちょっと浮いているような印象を受けます。

勝敗回数の表示

関数への分割が終了しました。今度は、ゲーム終了後に、《○勝○負○分けでした。》と表示する機能を追加しましょう。そのためには、勝った回数、負けた回数、引き分けた回数を格納するための変数が必要です。変更したプログラムを以下に示します。

■ じゃんけんゲーム（Ver.2.1）

```
/*
    じゃんけんゲーム（Ver.2.1：勝／負／引分けの回数を表示）
*/

#include  <ctime>
#include  <cstdlib>
#include  <iostream>

using namespace  std;
```

```
int  user;            // ユーザの手
int  comp;            // コンピュータの手

int  win_no;          // 勝った回数
int  lose_no;         // 負けた回数
int  draw_no;         // 引き分けた回数

char hd[][7] = {"グー", "チョキ", "パー"};

/*--- 初期処理 ---*/
void initialize(void)
{
    win_no = 0;           // 勝った回数
    lose_no = 0;          // 負けた回数
    draw_no = 0;          // 引き分けた回数
    time_t t;             // 時刻

    srand(time(&t) % RAND_MAX);      // 乱数の種を初期化

    cout << "じゃんけんゲーム開始。¥n";
}

/*--- じゃんけん実行 ---*/
void jyanken(void)
{
    /* 省略（Ver.2.0）と同じ */
}

/*--- 手を表示 ---*/
void disp_hands(void)
{
    /* 省略（Ver.2.0）と同じ */
}

/*--- 判定結果を表示 ---*/
void disp_result(int result)
{
    switch (result) {
     case 0: cout << "引き分けです。¥n";          // 引き分け
             draw_no++;
             break;

     case 1: cout << "あなたの負けです。¥n";      // 負け
             lose_no++;
             break;

     case 2: cout << "あなたの勝ちです。¥n";      // 勝ち
             win_no++;
             break;
    }
}
```

```
/*--- 再挑戦するかを確認 ---*/
int confirm_retry(void)
{
    /* 省略（Ver.2.0）と同じ */
}

int main(void)
{
    int  retry;                  // もう一度？

    initialize();                // 初期処理

    do {
        jyanken();               // じゃんけん実行

        disp_hands();            // コンピュータとユーザの手を表示

        int result = (user - comp + 3) % 3; // 判定

        disp_result(result);     // 判定結果を表示

        retry = confirm_retry(); // 再挑戦するかを確認

    } while (retry == 1);

    cout << win_no << "勝" << lose_no << "負" << draw_no << "分けでした。¥n";

    return (0);
}
```

■　回数を格納する変数 win_no, lose_no, draw_no を初期化するために、initialize という初期化用の関数を追加しました。

なお、乱数の種を初期化する機能も、この関数に入れたことによって、プログラムがすっきりしました。

■　判定結果を表示する関数 disp_result が変更されています。表示と同時に、勝ち／負け／引き分けの回数のいずれかがインクリメントしています。

■　main 関数の最後で、勝敗回数を表示しています。

さて、関数 **disp_result** は、いずれも仮引数 **diff** の値によって、勝ち／負け／引き分けを表示し、回数をカウントアップします（Ver.3 までは if 文でしたが、switch 文の方が読みやすいので変更しました)。

しかし、内部的な回数のカウントと、外部的な表示は、本質的にまったく異質の処理です。したがって、勝ち／負け／引き分けのメッセージを変更する際や、じゃんけんを三人にすることによって勝敗の判断や表示メッセージを変更する際は、このような関数の変更は大変になります。

したがって、関数のカウントとメッセージの表示は、それぞれ独立した関数として実現すべきであることが分かります。

以上の点を改良しましょう。

■ じゃんけんゲーム（Ver.2.2）

```
/*
    じゃんけんゲーム（Ver.2.2）
*/

（途中省略：Ver.2.1と同じ）

/*--- 初期処理 ---*/
void initialize(void)
{
    （詳細省略：Ver.2.1と同じ）
}

/*--- じゃんけん実行 ---*/
void jyanken(void)
{
    （詳細省略：Ver.2.1と同じ）
}

/*--- 手を表示 ---*/
void disp_hands(void)
{
    （詳細省略：Ver.2.1と同じ）
}

/*--- 勝／負／引分け回数を更新 ---*/
void count_no(int result)
{
    switch (result) {
     case 0: draw_no++;  break;      // 勝ち
     case 1: lose_no++;  break;      // 負け
     case 2: win_no++;     break;    // 引分け
    }
}
```

```
/*--- 判定結果を表示 ---*/
void disp_result(int result)
{
    switch (result) {
     case 0: cout << "引き分けです。¥n";       break;
     case 1: cout << "あなたの負けです。¥n";   break;
     case 2: cout << "あなたの勝ちです。¥n";   break;
    }
}

/*--- 再挑戦するかを確認 ---*/
int confirm_retry(void)
{
     （詳細省略：Ver.2.1と同じ）
}

int main(void)
{
    int  retry;                  // もう一度？

    initialize();                // 初期処理

    do {
        jyanken();               // じゃんけん実行

        disp_hands();            // コンピュータとユーザの手を表示

        int  result = (user - comp + 3) % 3;        // 判定

        count_no(result);        // 勝／負／引分け回数を更新

        disp_result(result);     // 判定結果を表示

        retry = confirm_retry(); // 再挑戦するかを確認
    } while (retry == 1);

    cout << win_no << "勝" << lose_no << "負" << draw_no << "分けでした。¥n";

    return (0);
}
```

■ ソースファイルの分割

関数としては、じゃんけんのゲームの本質部分と、表示や入力などの入出力関係などのインタフェース部分は独立して実現すべきであることが分かりました。

今後の仕様変更、機能追加を見越して、それらをきちんとまとめるとともに、ソースファイルを分割しましょう。ここでは、

- ■ 入出力関係　**jnkio.cpp**
- ■ 判定等関係　**jnkjd.cpp**
- ■ メイン部分　**jyanken.cpp**

の三つに分割します。それらのプログラムを示します。

■ じゃんけんゲーム Ver.3.0 ･･･････････････　**jnkio.cpp**：入出力関係

```
/*
    じゃんけんゲーム Ver.3.0（jnkio.cpp：入出力関係）
*/

#include  <iostream>

using namespace  std;

char hd[][7] = {"グー", "チョキ", "パー"};

/*--- 開始メッセージを表示 ---*/
void disp_start(void)
{
    cout << "じゃんけんゲーム開始。¥n";
}

/*--- ユーザの手を入力 ---*/
int get_user_hand(void)
{
    int  hand;            // 手

    cout << "¥n¥aじゃんけんポン … ";
    for (int i = 0; i < 3; i++)
        cout << "(" << i << ")" << hd[i] << " ";
    cout << "：";

    cin >> hand;          // ユーザの手を読み込む

    return (hand);
}

/*--- 手を表示 ---*/
void disp_hands(int comp, int user)
{
    cout << "私は"      << hd[comp] << "で、"
         << "あなたは"  << hd[user] << "です。¥n";
}
```

```
/*--- 判定結果を表示 ---*/
void disp_result(int result)
{
    switch (result) {
     case 0: cout << "引き分けです。¥n";       break;
     case 1: cout << "あなたの負けです。¥n";   break;
     case 2: cout << "あなたの勝ちです。¥n";   break;
    }
}

/*--- 再挑戦するかを確認 ---*/
int confirm_retry(void)
{
    int  x;
    cout << "もう一度しますか … (0)いいえ (1)はい：";
    cin >> x;

    return (x);
}

/*--- 最終結果を表示 ---*/
void disp_record(int win, int lose, int draw)
{
    cout << win << "勝" << lose << "負" << draw << "分けでした。¥n";
}
/*--- 手を表示 ---*/
void disp_hands(int comp, int user)
{
    cout << "私は"      << hd[comp] << "で、"
         << "あなたは"  << hd[user] << "です。¥n";
}

/*--- 判定結果を表示 ---*/
void disp_result(int result)
{
    switch (result) {
     case 0: cout << "引き分けです。¥n";       break;
     case 1: cout << "あなたの負けです。¥n";   break;
     case 2: cout << "あなたの勝ちです。¥n";   break;
    }
}

/*--- 再挑戦するかを確認 ---*/
int confirm_retry(void)
{
    int  x;
    cout << "もう一度しますか … (0)いいえ (1)はい：";
    cin >> x;

    return (x);
}
```

```
/*--- 最終結果を表示 ---*/
void disp_record(int win, int lose, int draw)
{
    cout << win << "勝" << lose << "負" << draw << "分けでした。¥n";
}
```

■ソースファイル〝jnkio.cpp〟には、表示や入力などの入出力関係がまとめられています。
ソースファイル〝jnkjd.cpp〟には、初期化や判定などの関数がまとめられています。
ソースファイル〝jyanken.cpp〟がメインとなる部分です。

■　〝グー〟, 〝チョキ〟, 〝パー〟を表す文字列の配列 hd は、表示の際にのみ必要であり、判定等では不要ですから、〝jnkio.cpp〟中で宣言され、このソースプログラム内でのみ利用されています。

■　入出力を行うために必要な<iostream>ヘッダは、〝jnkio.cpp〟からのみインクルードされており、他のソースプログラムからはインクルードされていません。

■　勝／負／引分けを格納する変数である win_no, lose_no, draw_no は〝jyanken.cpp〟で宣言・定義されています。〝jnkjd.cpp〟では、自分以外のソースファイルで定義された変数を利用するために、

```
extern int  win_no;
```

と extern 付きで宣言されていることに注意しましょう。

■　メインとなる次ページの〝jyanken.cpp〟では、他のソースファイルで定義された関数を呼び出すために、関数原型（プロトタイプ）宣言が、プログラム冒頭部で、ずらずらと並べられています。

■ じゃんけんゲーム Ver.3.0 ・・・・・・・・・・・・・・・ **jnkjd.cpp**：判定等関係

```
/*
   じゃんけんゲーム Ver.3.0（jnkjd.cpp：判定等関係）
*/

#include  <ctime>
#include  <cstdlib>

using namespace  std;

extern int  win_no;          // 勝った回数
extern int  lose_no;         // 負けた回数
extern int  draw_no;         // 引き分けた回数

/*--- 初期処理 ---*/
void initialize(void)
{
   time_t t;           // 時刻

   win_no = 0;
   lose_no = 0;
   draw_no = 0;

   srand(time(&t) % RAND_MAX);      // 乱数の種を初期化
}

/*--- コンピュータの手を生成 ---*/
int generate_comp_hand(void)
{
   return (rand() % 3);      // コンピュータの手（0〜2）を乱数で生成
}


/*--- 勝／負／引分け回数を更新 ---*/
void count_no(int result)
{
   switch (result) {
    case 0: draw_no++;  break;      // 勝ち
    case 1: lose_no++;  break;      // 負け
    case 2: win_no++;   break;      // 引分け
   }
}
```

■ じゃんけんゲーム Ver.3.0 ・・・・・・・・・・・・・・・ **jyanken.cpp**：メイン部分

```
/*
    じゃんけんゲーム Ver.3.0（jyanken.cpp：メイン）
*/

#include  <iostream>

using namespace  std;

int  win_no;          // 勝った回数
int  lose_no;         // 負けた回数
int  draw_no;         // 引き分けた回数

void initialize(void);              // 初期処理
int generate_comp_hand(void);       // コンピュータの手を生成
void count_no(int result);          // 勝／負／引分け回数を更新
void disp_record(int, int, int);// 最終結果を表示

void disp_start(void);              // 開始メッセージを表示
int get_user_hand(void);        // ユーザの手を入力
void disp_hands(int, int);          // 手を表示
void disp_result(int result);       // 判定結果を表示
int confirm_retry(void);        // 再挑戦するかを確認

int main(void)
{
    int  retry;                 // もう一度？

    initialize();               // 初期処理

    do {
        int  comp = generate_comp_hand();   // コンピュータの手を生成
        int  user = get_user_hand();     // ユーザの手を入力

        disp_hands(comp, user);             // コンピュータとユーザの手を表示

        int  judge = (user - comp + 3) % 3;

        count_no(judge);          // 勝／負／引分け回数を更新

        disp_result(judge);           // 判定結果を表示

        retry = confirm_retry(); // 再挑戦するかを確認
    } while (retry == 1);

    disp_record(win_no, lose_no, draw_no);  // 最終結果を表示

    return (0);
}
```

■ ヘッダファイルの作成

正弦を求める **sin** 関数を利用する際に、わざわざ

```
double sin(double);
```

と関数原型宣言を宣言するのは、不恰好ですし、間違いのもとです。通常は、

```
#include  <cmath>          // C言語であれば #include  <math.h>
```

と、ヘッダを取り込みます。

標準で提供されるライブラリを利用する場合だけでなく、自分でプログラムを開発する際も、そのソースファイルの外部から呼び出される関数に関しては、関数原型宣言を含むヘッダを用意しなければなりません。

また、win_no, lose_no, draw_no はセットになった変数ですから、構造体としてまとめるべきです。また、不用意に広域的な変数となっているのは好ましくありません。

以上の２点を改良したプログラムを以下に示します。

※ヘッダが追加されますので、ファイルは四つとなります。

■ じゃんけんゲーム Ver.3.1 ・・・・・・・・・・・・・・・・　**jyanken.h**：ヘッダ

```
/*
    じゃんけんゲーム Ver.3.1 ヘッダ
*/

//--- 勝敗の回数 ---//
typedef struct {
    int  win;          // 勝った回数
    int  lose;         // 負けた回数
    int  draw;         // 引き分けた回数
} record;

//--- jnkio.cpp：入出力関係 ---//
void initialize(record*);        // 初期処理
int generate_comp_hand(void);       // コンピュータの手を生成
void count_no(int result, record*); // 勝／負／引分け回数を更新
void disp_record(record);        // 最終結果を表示

//--- jnkjd.cpp：判定等関係 ---//
void disp_start(void);              // 開始メッセージを表示
int get_user_hand(void);         // ユーザの手を入力
void disp_hands(int, int);          // 手を表示
void disp_result(int result);       // 判定結果を表示
int confirm_retry(void);         // 再挑戦するかを確認
```

■ じゃんけんゲーム Ver.3.1 ・・・・・・・・・・・・・・・ **jnkio.cpp**：入出力関係

```
/*
    じゃんけんゲーム Ver.3.1（jnkio.cpp：入出力関係）
*/

#include  <iostream>
#include  "jyanken.h"

using namespace  std;

char hd[][7] = {"グー", "チョキ", "パー"};

/*--- 開始メッセージを表示 ---*/
void disp_start(void)
{
     （詳細省略：Ver.3.0と同じ）
}

/*--- ユーザの手を入力 ---*/
int get_user_hand(void)
{
     （詳細省略：Ver. 3.0と同じ）
}

/*--- 手を表示 ---*/
void disp_hands(int comp, int user)
{
     （詳細省略：Ver. 3.0と同じ）
}

/*--- 判定結果を表示 ---*/
void disp_result(int result)
{
     （詳細省略：Ver. 3.0と同じ）}

/*--- 再挑戦するかを確認 ---*/
int confirm_retry(void)
{
     （詳細省略：Ver. 3.0と同じ）
}

/*--- 最終結果を表示 ---*/
void disp_record(record rec)
{
    cout << rec.win << "勝" << rec.lose << "負" << rec.draw << "分けでした。¥n";
}
```

■　jyanken.h 内で宣言されている関数原型宣言との整合性をとるためにも、そのヘッダをインクルードします（以降でも同様です）。

■ じゃんけんゲーム Ver.3.1 ・・・・・・・・・・・・・・・ **jnkjd.cpp**：判定等関係

```
/*
    じゃんけんゲーム Ver.3.1（jnkjd.cpp：判定等関係）
*/

#include  <ctime>
#include  <cstdlib>
#include  "jyanken.h"

using namespace   std;

/*--- 初期処理 ---*/
void initialize(record *rec)
{
    time_t t;             // 時刻

    rec->win  = 0;
    rec->lose = 0;
    rec->draw = 0;

    srand(time(&t) % RAND_MAX);      // 乱数の種を初期化
}

/*--- コンピュータの手を生成 ---*/
int generate_comp_hand(void)
{
    return (rand() % 3);             // コンピュータの手（0〜2）を乱数で生成
}

/*--- 勝／負／引分け回数を更新 ---*/
void count_no(int result, record *rec)
{
    switch (result) {
     case 0: rec->draw++;  break;    // 引分け
     case 1: rec->lose++;  break;    // 負け
     case 2: rec->win++;   break;    // 勝ち
    }
}
```

■　勝敗回数を記憶させる目の構造体 record 型の導入に伴って、関数 initialize や cout_no が変更されていることに注意しましょう。

■ じゃんけんゲーム Ver.3.1 ・・・・・・・・・・・・・・・ **jyanken.cpp**：メイン部分

```
/*
    じゃんけんゲーム Ver.3.1（jyanken.cpp：メイン）
*/

#include  <iostream>
#include  "jyanken.h"

using namespace  std;

int main(void)
{
    int  retry;                     // もう一度？
    record rec;                     // 勝／負／引分け回数

    initialize(&rec);               // 初期処理

    do {
        int  comp = generate_comp_hand();   // コンピュータの手を生成
        int  user = get_user_hand();        // ユーザの手を入力

        disp_hands(comp, user);             // コンピュータとユーザの手を表示

        int  judge = (user - comp + 3) % 3;

        count_no(judge, &rec);      // 勝／負／引分け回数を更新

        disp_result(judge);         // 判定結果を表示

        retry = confirm_retry();    // 再挑戦するかを確認
    } while (retry == 1);

    disp_record(rec);               // 最終結果を表示

    return (0);
}
```

■ ３人じゃんけんへの拡張

じゃんけんプログラムのプレーヤーを３人にしましょう。ここでは、コンピュータを２人とし、それらの手は乱数によって生成するものとします。プログラムを示します。

■ じゃんけんゲーム Ver.4 ・・・・・・・・・・・・・・・　**jyanken.h**：ヘッダ

```
/*
    じゃんけんゲーム Ver.4 ヘッダ
*/

//--- 勝敗の回数 ---//
typedef struct {
    int  win;          // 勝った回数
    int  lose;         // 負けた回数
    int  draw;         // 引き分けた回数
} record;

//--- jnkio.cpp：入出力関係 ---//
void initialize(record*, record*, record*);            // 初期処理
int generate_comp_hand(void);                          // コンピュータの手を生成
void judge(int, int, int, int*, int*, int*);           // 判定
void count_no(int, int, int, record*, record*, record*);// 勝／負／引分け回数を更新
void disp_record(record, record, record);              // 最終結果を表示

//--- jnkjd.cpp：判定等関係 ---//
void disp_start(void);              // 開始メッセージを表示
int get_user_hand(void);         // ユーザの手を入力
void disp_hands(int, int, int);     // 手を表示
void disp_result(int, int, int);// 判定結果を表示
int confirm_retry(void);         // 再挑戦するかを確認
```

■ じゃんけんゲーム Ver.4 ・・・・・・・・・・・・・・・　**jnkio.cpp**：入出力関係

```
/*
    じゃんけんゲーム Ver.4（jnkio.cpp：入出力関係）
*/

#include  <iostream>
#include  "jyanken.h"

using namespace  std;

char hd[][7] = {"グー", "チョキ", "パー"};

/*--- 開始メッセージを表示 ---*/
void disp_start(void)
{
    cout << "じゃんけんゲーム開始。¥n";
}

/*--- ユーザの手を入力 ---*/
int get_user_hand(void)
{
```

```
    （詳細省略：Ver. 3.0と同じ）}
}

/*--- 手を表示 ---*/
void disp_hands(int user, int cmp1, int cmp2)
{
    cout << "あなたは《" << hd[user] << "》で"
         << "Comp１は《"  << hd[cmp1] << "》で"
         << "Comp２は《"  << hd[cmp2] << "》です。¥n";
}

/*--- 判定結果を表示 ---*/
void disp_result(int juser, int jcmp1, int jcmp2)
{
    if (juser == 2) {
        if (jcmp1 == 2)
            cout << "あなたとComp1の勝ちです。¥n";
        else if (jcmp2 == 2)
            cout << "あなたとComp2の勝ちです。¥n";
        else
            cout << "あなたの勝ちです。¥n";
    } else {
        if (jcmp1 == 2)
            if (jcmp2 == 2)
                cout << "Comp1とComp2の勝ちです。¥n";
            else
                cout << "Comp1の勝ちです。¥n";
        else if (jcmp2 == 2)
            cout << "Comp2の勝ちです。¥n";
        else
            cout << "引き分けです。¥n";
    }
}

/*--- 再挑戦するかを確認 ---*/
int confirm_retry(void)
{
    （詳細省略：Ver. 3.0と同じ）}
}

/*--- 最終結果を表示 ---*/
void disp_record(record ruser, record rcmp1, record rcmp2)
{
    cout << "あなた：" << ruser.win << "勝" << ruser.lose << "負" << ruser.draw << "
分けでした。¥n";
    cout << "Comp１：" << rcmp1.win << "勝" << rcmp1.lose << "負" << rcmp1.draw << "
分けでした。¥n";
    cout << "Comp２：" << rcmp2.win << "勝" << rcmp2.lose << "負" << rcmp2.draw << "
分けでした。¥n";
}
```

■ じゃんけんゲーム Ver.5 ・・・・・・・・・・・・・・・ **jnkjd.cpp**：判定等関係

```
/*
    じゃんけんゲーム Ver.4（jnkio.cpp：入出力関係）
*/

#include  <iostream>
#include  "jyanken.h"

using namespace  std;

char hd[][7] = {"グー", "チョキ", "パー"};

/*--- 開始メッセージを表示 ---*/
void disp_start(void)
{
    cout << "じゃんけんゲーム開始。¥n";
}

/*--- ユーザの手を入力 ---*/
int get_user_hand(void)
{
    int  hand;            // 手

    cout << "¥n¥aじゃんけんポン … ";
    for (int i = 0; i < 3; i++)
        cout << "(" << i << ")" << hd[i] << " ";
    cout << "：";

    cin >> hand;          // ユーザの手を読み込む

    return (hand);
}

/*--- 手を表示 ---*/
void disp_hands(int user, int cmp1, int cmp2)
{
    cout << "あなたは《" << hd[user] << "》で"
         << "Comp１は《"  << hd[cmp1] << "》で"
         << "Comp２は《"  << hd[cmp2] << "》です。¥n";
}

/*--- 判定結果を表示 ---*/
void disp_result(int juser, int jcmp1, int jcmp2)
{
    if (juser == 2) {
        if (jcmp1 == 2)
            cout << "あなたとComp1の勝ちです。¥n";
        else if (jcmp2 == 2)
            cout << "あなたとComp2の勝ちです。¥n";
        else
            cout << "あなたの勝ちです。¥n";
    } else {
```

```
        if (jcmp1 == 2)
            if (jcmp2 == 2)
                cout << "Comp1とComp2の勝ちです。¥n";
            else
                cout << "Comp1の勝ちです。¥n";
        else if (jcmp2 == 2)
            cout << "Comp2の勝ちです。¥n";
        else
            cout << "引き分けです。¥n";
    }
}

/*--- 再挑戦するかを確認 ---*/
int confirm_retry(void)
{
    int  x;
    cout << "もう一度しますか … (0)いいえ (1)はい：";
    cin >> x;

    return (x);
}

/*--- 最終結果を表示 ---*/
void disp_record(record ruser, record rcmp1, record rcmp2)
{
    cout << "あなた：" << ruser.win << "勝" << ruser.lose << "負"
                                    << ruser.draw << "分けでした。¥n";
    cout << "Comp１：" << rcmp1.win << "勝" << rcmp1.lose << "負"
                                     << rcmp1.draw << "分けでした。¥n";
    cout << "Comp２：" << rcmp2.win << "勝" << rcmp2.lose << "負"
                                      << rcmp2.draw << "分けでした。¥n";
}
```

２人のじゃんけんでは、利用者（人間）の勝／負／引分けの回数のみを記憶すれば十分でした（コンピュータの回数は、それから計算できますから）が、３人のじゃんけんでは、全員の回数を記憶させなければならないことに注意しましょう。したがって、そのための変数、それに伴う処理が増えています。

■ じゃんけんゲーム Ver.4 ・・・・・・・・・・・・・・・ **jyanken.cpp**：メイン部分

```
/*
    じゃんけんゲーム Ver.4（３人じゃんけん）
*/

#include  "jyanken.h"

int main(void)
{
    int  retry;                 // もう一度？
    record rec_user;            // 勝／負／引分け回数（ユーザ）
    record rec_comp1;           //       〃          （コンピュータ１）

    record rec_comp2;           //       〃          （コンピュータ２）


    disp_start();               // 開始メッセージを表示

    initialize(&rec_user, &rec_comp1, &rec_comp2); // 初期処理

    do {
        int  comp1 = generate_comp_hand();  // コンピュータの手を生成
        int  comp2 = generate_comp_hand();  // コンピュータの手を生成
        int  user  = get_user_hand();    // ユーザの手を入力
        int  jdcmp1, jdcmp2, jduser;     // コンピュータとユーザの勝／負け／引分け

        disp_hands(user, comp1, comp2);     // コンピュータとユーザの手を表示

        judge(user, comp1, comp2, &jduser, &jdcmp1, &jdcmp2);

        count_no(jduser, jdcmp1, jdcmp2, &rec_user, &rec_comp1, &rec_comp2);

        disp_result(jduser, jdcmp1, jdcmp2);       // 判定結果を表示

        retry = confirm_retry();               // 再挑戦するかを確認
    } while (retry == 1);

    disp_record(rec_user, rec_comp1, rec_comp2);   // 最終結果を表示

    return (0);
}
```

メインとなるプログラムである jyanken.cpp だけをパッと見ると、プログラムの改良が小規模に押さえられているように感じられます。

しかし、いくつかの関数の仕様は、微妙に変更されているため、２人じゃんけんとは**互換性が保てない**ものとなっています。

また、各関数の引数が多いことも気になります。一概には言えませんが、引数の多い関数というのは、処理の単位として、あるいは入出力データの関係が、**きちんとまとまっていない**ことを表します。

クラス

じゃんけんゲームのプレーヤー（コンピュータ／人間)、じゃんけんゲームなどは、クラスとして実現した方が、よりスマートになります。

■プレーヤーのデータについて

コンピュータでも人間でも、プレーヤーには、

□現在の手（グー、チョキ、パー）

□勝敗数

・勝ち数

・負け数

・引き分け数

のデータがあります。さらなる仕様の変更では、プレーヤー名を表す文字列なども追加される可能性があります。

プログラムでは、プレーヤーの手を表す変数 **user**，**comp1**，**comp2** と、勝敗数を表す変数 **rec_user**，**rec_comp1**，**rec_cmp2** とが、ばらばらに宣言・定義されています。変数名から、それらの関係を推測することはできても、100%確定することはできません。

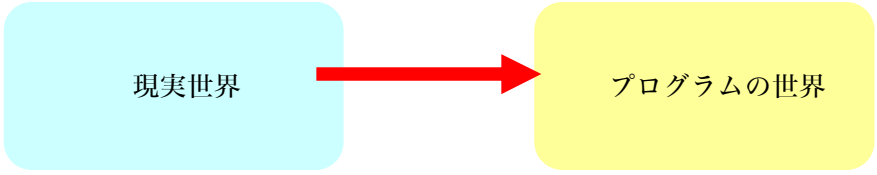
このように、バラバラな命名法を続けていったら、プログラムは、たくさんのオブジェクトの名前で溢れかえってしまいます。プレーヤーは４人、５人となるかもしれません。

■オブジェクトとクラス

私達人間が出す『手』を表すために、**user** というオブジェクト（変数）を使っているわけですが、これは、現実世界のオブジェクト（物）を、プログラムの世界のオブジェクト（変数）に投影した結果の産物なのです。

人間の一つ側面のみに着目するのでなく、複数の側面に着目し、「手」、「勝ち数」、「負け数」、必要ならば「プレーヤー名」をセットにした投影を行うのがクラスの考え方です。

プログラムで扱う問題の種類や範囲によっても異なりますが、現実の世界からプログラムの世界への投影は、『まとめるべきものはまとめる』といった方針に則った方が、より自然で素直なものとなります。



■クラスの定義

ここでは、ちょっとじゃんけんプログラムを離れて、車のクラスを考えていきましょう。
プレーヤーを表すクラスの試作品を作ってみましょう。以下のプログラムです。

■ 車クラス Ver.0.1 ・・・・・・・・・・・・・・・ **car0001.cpp**

武田君の Vitz 号のスペック：
長さ 3640mm で幅 1660mm で高さ 1500mm でタンク 40 ﾘｯﾄﾙです。

立野君のマーちゃんのスペック：
長さ 3695mm で幅 1660mm で高さ 1525mm でタンク 41 ﾘｯﾄﾙです。

```
/*
    車クラス（Ver.0.1）
*/

#include  <iostream>

using namespace std;

class Car {
public:
    string name;      // 車名
    int    length;    // 車長
    int    width;     // 車幅
    int    height;    // 車高
    double tank;      // ガソリンタンク容量
    double gasolin;   // 現在のガソリン量
};

int main(void)
{
    Car  Vitz;        // 武田君のVitz
    Car  March;       // 立野君のMarch

    Vitz.name    = "武田君のVitz号";
    Vitz.length  = 3640;
    Vitz.width   = 1660;
    Vitz.height  = 1500;
    Vitz.tank    = 40.0;
    Vitz.gasolin = 40.0;

    March.name    = "立野君のマーちゃん";
    March.length  = 3695;
    March.width   = 1660;
    March.height  = 1525;
    March.tank    = 41.0;
    March.gasolin = 0.0;

    cout << Vitz.name << "のスペック：¥n"
         << "長さ"   << Vitz.length << "mmで"
         << "幅"     << Vitz.width  << "mmで"
         << "高さ"   << Vitz.height << "mmで"
         << "タンク" << Vitz.tank   << "ﾘｯﾄﾙです。¥n";

    cout << "¥n";
```

```
    cout << March.name << "のスペック：¥n"
         << "長さ"   << March.length << "mmで"
         << "幅"     << March.width  << "mmで"
         << "高さ"   << March.height << "mmで"
         << "タンク" << March.tank   << "ﾘｯﾄﾙです。¥n";

    return (0);
}
```

■クラスの宣言

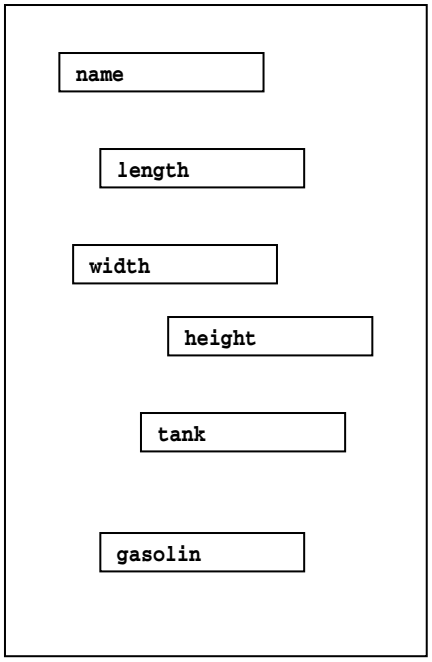
先頭の **class Car {** は、これから **Car** という名前のクラスを宣言することを表します。そして、その宣言は};まで続くことになります。

クラス宣言の中では、

■データメンバ（*data member*）

■メンバ関数（*member function*）

などを宣言することができます。このクラスでは、六つのデータメンバがあり、メンバ関数はありません。



クラス Car は、『車とはこういうものだよ。』と、車の性質を与えるものであって、実体ではありません。

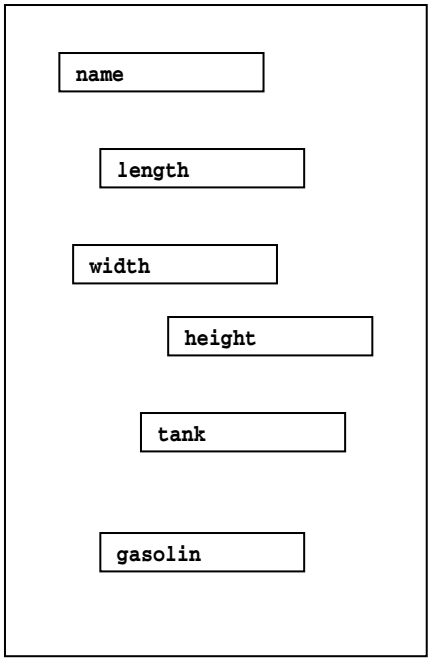
■アクセス制御

クラス先頭の **public:** は、それ以降に宣言するメンバを、クラスの外部に対して公開することを意味します。もし、公開しなければ、クラスの外からは、そのメンバの名前や型はおろか、その存在を知ることすらできません。

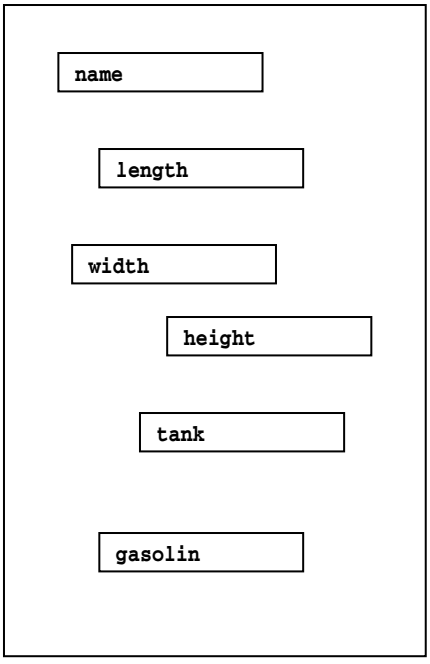
なお、非公開を明示的に指示するには **private:**を挿入します。この他にも **protected:** があり、これらはクラス宣言内で任意の順序で交互に指定することができます。

さて、このプログラムは以下のように感じられます。

■　Vits と March に対して、似たような部分が繰り返されています。
もっとスマートに初期化できればいいはずです。



Vits



March

■　現在のガソリン量は、給油や走行によって変化しますが、それ以外の長さや幅などは、原則として変化しないはずのものです。それらは公開しないほうがいいはずです。
すなわち、クラスの外部から勝手に値をいじくれないようにすべきです。

書き換えたプログラムを次ページに示します。

```
/*
    車クラス（Ver.0.2）
*/

#include  <iostream>

using namespace std;

class Car {
    string name;      // 車名
    int    length;    // 車長
    int    width;     // 車幅
    int    height;    // 車高
    double tank;      // ガソリンタンク容量
public:
    double gasolin;   // 現在のガソリン量

    // コンストラクタ
    Car(string n, int l, int w, int h, double t, double g)
    {
        name    = n;  // 車名
        length  = l;  // 車長
        width   = w;  // 車幅
        height  = h;  // 車高
        tank    = t;  // ガソリンタンク容量
        gasolin = g;  // 現在のガソリン量
    }

    // スペック表示
    void PutSpec()
    {
        cout << name << "のスペック：¥n"
             << "長さ"   << length << "mmで"
             << "幅"     << width  << "mmで"
             << "高さ"   << height << "mmで"
             << "タンク" << tank   << "ﾘｯﾄﾙです。¥n";
        }
};

int main(void)
{
    Car  Vitz("武田君のVitz号", 3640, 1660, 1500, 40.0, 40.0);
    Car  March("立野君のマーちゃん", 3695, 1660, 1525, 41.0, 0.0);

    Vitz.PutSpec();

    cout << "¥n";

    March.PutSpec();

    return (0);
}
```

■公開部と私的部

新しいクラスでは、

・車名
・車長　　私的
・車幅
・車高

を私的にして、クラスの外からアクセスできないようにしています。また、

・ガソリンタンク容量　　公開

を公開して、クラスの外からアクセスできるようにしています。

○○　演習　○○

main 関数で、

```
vitz.length = 4000;
```

あるいは

```
vitz.gasolin = 30.0;
```

と行うとどうなるか。確認せよ。

■コンストラクタ

クラスと同じ名前をもつメンバ関数のことを、コンストラクタ（*constructor*）と呼びます。コンストラクタは、そのクラス型のオブジェクトが生成する際に呼び出される関数です。

クラス **Car** のコンストラクタは、

・車名
・車長
・車幅
・車高
・ガソリンタンク容量
・現在のガソリン量

を受け取って、その値をデータメンバにセットします。

したがって、**Ver.0.1** のプログラムのように、Vits と March の各メンバを個々に初期化するコードを記述する必要から解放されます。

■スペックを表示するメンバ関数

メンバ関数 **PutSpec** は、車のスペックを表示する関数です。

さて、Vitz はガソリンが満タンですが、March は空っぽです。一般に、車は初期状態ではタンクのガソリンは空です。プログラムでのクラス Car 型の変数を初期化する際は、必ずしもガソリン量まで指定する必要はないでしょう。

したがって、ガソリン量を指定できるようにし、指定しなければ自動的に 0 になるようにするとよさそうです。

さらに、ガソリンの給油の関数を追加しましょう。プログラムは次のようになります。

■ 車クラス Ver.0.3 ・・・・・・・・・・・・・・・・ **car0003.cpp**

```
/*
   車クラス（Ver.0.3）
*/

#include  <iostream>

using namespace std;

class Car {
   string name;      // 車名
   int    length;    // 車長
   int    width;     // 車幅
   int    height;    // 車高
   double tank;      // ガソリンタンク容量
public:
   double gasolin;   // 現在のガソリン量

   // コンストラクタ
   Car(string n, int l, int w, int h, double t, double g = 0)
   {
       name    = n;  // 車名
       length  = l;  // 車長
       width   = w;  // 車幅
       height  = h;  // 車高
       tank    = t;  // ガソリンタンク容量
       gasolin = g;  // 現在のガソリン量
   }

   // スペック表示
   void PutSpec()
   {
       cout << name << "のスペック：¥n"
            << "長さ"   << length << "mmで"
            << "幅"     << width  << "mmで"
            << "高さ"   << height << "mmで"
            << "タンク" << tank   << "ﾘｯﾄﾙです。¥n";
   }

   // 給油
   void Kyuyu(double gas)
   {
```

```
        gasolin += gas;
    }
};

int main(void)
{
    Car  Vitz("武田君のVitz号", 3640, 1660, 1500, 40.0, 40.0);
    Car  March("立野君のマーちゃん", 3695, 1660, 1525, 41.0);


    March.Kyuyu(13.5);        // 13.5ﾘｯﾄﾙ給油

    Vitz.PutSpec();
    cout << "現在のガソリンが" << Vitz.gasolin << "ﾘｯﾄﾙあります。¥n";

    cout << "¥n";

    March.PutSpec();
    cout << "現在のガソリンが" << March.gasolin << "ﾘｯﾄﾙあります。¥n";

    return (0);
}
```

★　コンストラクタの最後の引数であるガソリン量には、引数のデフォルト値 0 が与えられています。したがって、

```
Car  Vitz("武田君のVitz号",      3640, 1660, 1500, 40.0, 40.0);
Car  March("立野君のマーちゃん", 3695, 1660, 1525, 41.0);
```

と宣言された March のガソリン量は 0 となります。

★　main 関数では、March に給油するために、

```
March.Kyuyu(13.5);
```

と行っていますが、これは、

```
March.gasolin += 13.5;
```

でも可能です。

このことから、メンバ gasolin も私的メンバとすべきであることが分かります。

その場合、ガソリンの量を知るためには、その値を返すメンバ関数を追加しなければなりません。

★　さて、ガソリンのタンク容量を超える給油が行われないようにしなければなりません。

★　走行した場合は、ガソリンが減るはずですね。

また、ガソリン量が空になるまでしか走れません（マイナスになることはありません）。

燃費を 1 として、クラスを改良しましょう。

■ 車クラス Ver.0.4 ・・・・・・・・・・・・・・・ **car0004.cpp**

```
/*
    車クラス（Ver.0.4）
*/

#include  <iostream>

using namespace std;

class Car {
    string name;      // 車名
    int    length;    // 車長
    int    width;     // 車幅
    int    height;    // 車高
    double tank;      // ガソリンタンク容量
    double gasolin;   // 現在のガソリン量

public:

    // コンストラクタ
    Car(string n, int l, int w, int h, double t, double g = 0)
    {
        name    = n;  // 車名
        length  = l;  // 車長
        width   = w;  // 車幅
        height  = h;  // 車高
        tank    = t;  // ガソリンタンク容量
        gasolin = g;  // 現在のガソリン量
    }

    // 現在のガソリン量を返す
    double fuel(void)
    {
        return (gasolin);
    }

    // スペック表示
    void PutSpec()
    {
        cout << name << "のスペック：¥n"
             << "長さ"   << length << "mmで"
             << "幅"     << width  << "mmで"
             << "高さ"   << height << "mmで"
              << "タンク" << tank   << "㍑です。¥n";
    }

    // 給油
    void Kyuyu(double gas)
    {
        gasolin += gas;
        if (gasolin > tank)
            gasolin = tank;
```

```
    }

    // 走行（燃費は1とする）
    void Run(double x)
    {
        gasolin -= x;
        if (gasolin < 0)
            gasolin = 0;
    }
};

int main(void)
{
    Car  Vitz("武田君のVitz号", 3640, 1660, 1500, 40.0, 40.0);
    Car  March("立野君のマーちゃん", 3695, 1660, 1525, 41.0);

    March.Kyuyu(13.5);        // 13.5ﾘｯﾄﾙ給油

    Vitz.Run(20.0);           // 20.0ｷﾛ走行
    March.Run(8.7);           //  8.7ｷﾛ走行

    Vitz.PutSpec();
    cout << "現在のガソリンが" << Vitz.fuel() << "ﾘｯﾄﾙあります。¥n";

    cout << "¥n";

    March.PutSpec();
    cout << "現在のガソリンが" << March.fuel() << "ﾘｯﾄﾙあります。¥n";

    return (0);
}
```

★　メンバ gasolin は、私的メンバとなりしたので、

```
vitz.gasolin = 30.0;
```

とできなくなりました。

★　メンバ関数 fuel は、現在のガソリン量をそのまま返すだけの関数です。

★　給油を行う関数 Kyuyu は、ガソリン量がタンクを超えた場合は、タンク量とします。

★　走行を行う関数 Run は、ガソリンを消費します。ガソリン量が 0 を下回る場合は、0 とします。

日付クラスを作りましょう。

■ 日付クラス・インタフェース部

```
//-------------------------------------------------------------------------//
//  日付クラス Date 演算子版（インタフェース部）          "date.h"      //
//-------------------------------------------------------------------------//

#include   <iostream.h>

//===== 日付クラス =====//
class Date {

private:
    int    year;      // 西暦年
    int    month;     // 月
    int    day;       // 日

    static int Date::mday[12];      // 各月の日数 {31, 28, 31, 30, ··· }
    static int isLeap(int year);           // year年は閏年か？ (0 / 1)
    static int Ydays(int year);               // year年の日数（365 / 366）
    static int Mdays(int year, int month);    // year年month月の日数 (28～31)
    static int Dayof(int y, int m, int d);    // 元旦からの経過日数

public:
    Date(void);                               // デフォルトコンストラクタ
    Date(int y, int m = 1, int d = 1);        // コンストラクタ

    int Year(void)  const { return (year); }      // 年を返す
    int Month(void) const { return (month); }     // 月を返す
    int Day(void)   const { return (day); }       // 日を返す

    Date& operator++(void);                       // 1日進める
    Date& operator--(void);                       // 1日もどす
    Date operator++(int dmy);                 // 1日進める
    Date operator--(int dmy);                 // 1日もどす
    Date& operator+=(int dn);                 // dn日進める
    Date& operator-=(int dn);                 // dn日もどす
    friend Date operator+(const Date& date, int dn);      // dn日後を求める
    friend Date operator-(const Date& date, int dn);      // dn日前を求める
    friend long operator-(const Date& d1, const Date& d2);    // 日付の差を求める
    friend int operator==(const Date& d1, const Date& d2);    // 同じ日か？
    friend int operator!=(const Date& d1, const Date& d2);    // 違う日か？
    friend int operator>(const Date& d1, const Date& d2); // d1 ＞ d2
    friend int operator>=(const Date& d1, const Date& d2);    // d1 ≧ d2
    friend int operator<(const Date& d1, const Date& d2); // d1 ＜ d2
    friend int operator<=(const Date& d1, const Date& d2);    // d1 ≦ d2
};

ostream& operator<<(ostream&, const Date& x);             // 挿入演算子
```

■ 日付クラスヘッダ・実現部

```
//-------------------------------------------------------------------------//
//  日付クラス Date 演算子版（実現部）                    "date.c"      //
//-------------------------------------------------------------------------//

#include   <time.h>
#include   "date.h"

//--- 各月の日数 ---//
int Date::mday[12] = { 31, 28, 31, 30, 31, 30, 31, 31, 30, 31, 30, 31};

//--- year年は閏年か？（0･･･非閏年 / 1･･･閏年）---//
int Date::isLeap(int year)
{
    return (year % 4 == 0 && year % 100 != 0 || year % 400 == 0);
}

//--- year年の日数（365･･･非閏年 / 366･･･閏年）---//
int Date::Ydays(int year)
{
    return (365 + isLeap(year));
}

//--- year年month月の日数（28〜31）---//
int Date::Mdays(int year, int month)
{
    if (month-- != 2)                    // monthが２月でないとき
        return (mday[month]);
    return (mday[month] + isLeap(year));    // monthが２月であるとき
}

//--- 元旦からの経過日数 ---//
int Date::Dayof(int year, int month, int day)
{
    for (int m = 1; m < month; m++)
        day += Mdays(year, m);
    return (day);
}

//----- クラスDateのコンストラクタ -----//
Date::Date(int y, int m, int d)
{
    year  = y;
    month = m;
    day   = d;
}

//----- クラスDateのデフォルトコンストラクタ -----//
Date::Date(void){
    time_t     current;
    struct tm* local;
```

```
    time(&current);                     // 現在時刻を取得
    local = localtime(&current);    // 構造体に変換
    year  = local->tm_year + 1900;      // 年：tm_yearは年-1900
    month = local->tm_mon + 1;          // 月：tm_monは0〜11
    day   = local->tm_mday;
}

//--- 日付をdn日進める ( Date += int ) ---//
Date& Date::operator+=(int dn)
{
    if (dn < 0)
        return ((*this) -= -dn);

    day += dn;

    while (day > Mdays(year, month)) {
        day -= Mdays(year, month);
        if (++month > 12) {
            year++;
            month = 1;
        }
    }

    return (*this);
}

//--- 日付をdn日前を求める ( Date -= int ) ---//
Date& Date::operator-=(int dn)
{
    if (dn < 0)
        return ((*this) += -dn);

    day -= dn;

    while (day < 1) {
        if (--month < 1) {
            year--;
            month = 12;
        }
        day += Mdays(year, month);
    }

    return (*this);
}

//--- dateのdn日後を求める ( Date + int ) ---//
Date operator+(const Date& date, int dn)
{
    Date temp = date;
    return (temp += dn);
}

//--- dateのdn日前を求める ( Date - int ) ---//
```

```
Date operator-(const Date& date, int dn)
{
    Date temp = date;
    return (temp -= dn);
}

//--- 日付の差 d1 - d2 を求める ( Date - Date ) ---//
long operator-(const Date& d1, const Date& d2)
{
    long count = 0;
    long count1 = Date::Dayof(d1.year, d1.month, d1.day);
    long count2 = Date::Dayof(d2.year, d2.month, d2.day);

    if (d1.year == d2.year)
        count = count1 - count2;
    else if (d1.year > d2.year) {
        count = Date::Ydays(d2.year) - count2 + count1;
        for (int y = d2.year + 1; y < d1.year; y++)
            count += Date::Ydays(y);
    } else {
        count = -(Date::Ydays(d1.year) - count1 + count2);
        for (int y = d1.year + 1; y < d2.year; y++)
            count -= Date::Ydays(y);
    }

    return (count);
}

//--- １日進める ( ++Date ) ---//
Date& Date::operator++(void)
{
    if (day < Mdays(year, month))
        day++;
    else {
        if (++month >=  12) {
            year++;
            month = 1;
        }
        day = 1;
    }

    return (*this);
}

//--- １日進める ( Date++ ) ---//
Date Date::operator++(int n)
{
    Date temp = *this;
    ++(*this);
    return (temp);
}

//--- １日もどす ( --Date ) ---//
```

```
Date& Date::operator--(void)
{
    if (day >= 2)
        day--;
    else {
        if (--month <=1) {
            year--;
            month = 12;
        }
        day = Mdays(year, month);
    }

    return (*this);
}

//--- １日もどす ( Date-- ) ---//
Date Date::operator--(int n)
{
    Date temp = *this;
    --(*this);
    return (temp);
}

//--- d1とd2は同じ日か？ ( d1 == d2 ) ---//
int operator==(const Date& d1, const Date& d2)
{
    return (d1.year == d2.year  &&  d1.month == d2.month
                               &&  d1.day == d2.day);
}

//--- d1とd2は違う日か？ ( d1 != d2 ) ---//
int operator!=(const Date& d1, const Date& d2)
{
    return ( !(d1 == d2) );
}

//--- 日付の比較：d1 ＞ d2か？ ( d1 > d2 ) ---//
int operator>(const Date& d1, const Date& d2)
{
    if (d1.year  > d2.year) return (1);     // 年が異なる
    if (d1.year  < d2.year) return (0);     //     〃

    if (d1.month > d2.month) return (1);    // 年が等しい　ー 月が異なる
    if (d1.month < d2.month) return (0);    //     〃

    return (d1.day > d2.day);           //     〃       ー 月も等しい
}

//--- 日付の比較：d1 ＜ d2か？ ( d1 < d2 ) ---//
int operator<(const Date& d1, const Date& d2)
{
    if (d1.year  < d2.year) return (1);     // 年が異なる
    if (d1.year  > d2.year) return (0);     //     〃
```

```
    if (d1.month < d2.month) return (1);    // 年が等しい　ー 月が異なる
    if (d1.month > d2.month) return (0);    //     〃

    return (d1.day < d2.day);           //             ー 月も等しい
}

//--- 日付の比較：d1 ≧ d2か？ ( d1 >= d2 ) ---//
int operator>=(const Date& d1, const Date& d2)
{
    return (d1 > d2 || d1 == d2);
}

//--- 日付の比較：d1 ≦ d2か？ ( d1 <= d2 ) ---//
int operator<=(const Date& d1, const Date& d2){
    return (d1 < d2 || d1 == d2);
}

//--- 挿入演算子 ---//
ostream& operator<<(ostream &s, const Date& x)
{
    int  y, m, dw;
    char *dws[] = {"日", "月", "火", "水", "木", "金", "土"};

    y = x.Year();
    m = x.Month();
    if (m == 1  ||  m == 2) {
        y--;
        m += 12;
    }
    dw = (y + y/4 - y/100 + y/400 + (13*m+8)/5 + x.Day())  % 7;

    return (s << x.Year() << "年" << x.Month() << "月" << x.Day() << "日 (" << dws[dw]
<< ") ");
}
```

■日付クラスをテストするプログラム

```
//-------------------------------------------------------------------------//
//  日付クラス Date の利用例                          "testdate.c"     //
//-------------------------------------------------------------------------//

#include  "date.h"

int main(void)
{
    Date today;                       // 今日
    Date Birthday(1963, 11, 18);      // 誰かの誕生日

    cout << "今　日：" << today << '¥n';

    cout << "誕生日：" << Birthday << '¥n';

    cout << "生まれてから" << today - Birthday << "日経過しています。¥n";


    int y, m, d;
    cout << "年：";  cin >> y;
    cout << "月：";  cin >> m;
    cout << "日：";  cin >> d;

    Date xday(y, m, d);

    if (xday < today)
        cout << "それは今日より前です。¥n";
    else if (xday > today)
        cout << "それは今日より後です。¥n";
    else
        cout << "それは今日です。¥n";

    cout << "その日の10日後：" << xday + 10 << '¥n';
    cout << "その日の10日前：" << xday + 10 << '¥n';

    cout << "誕生日の" << xday - Birthday << "日後です。¥n";


    return (0);
}
```

■ 演習

車クラスに、購入日を追加しましょう。

```
/*
    クラスを用いたじゃんけんプログラム（Ver.1）
*/

#include  <string>
#include  <cstdlib>
#include  <iostream>

using namespace std;

class Player {
    int    type;      // 0…人間／1…コンピュータ
    string name;      // 名前
    int    hand;      // 手
    int    win;       // 勝ち数
    int    lose;      // 負け数
    int    draw;      // 引分け

public:
    Player(int t, string n)          // コンストラクタ
    {
        type = t;
        name = n;
        win = lose = draw = 0;
    }

    int newHand(void)                // 次の手
    {
        if (type == 0) {                 // ●人間
            cin >> hand;                 //     キーボードから読み込む
            return (hand);
        } else {                         // ●コンピュータ
            hand = rand() % 3;           //     乱数発生
            return (hand);
        }
    }

    void recordCount(int result)     // 勝敗回数を更新
    {
        switch (result) {
         case 0: draw++;  break;         // 引分け
         case 1: lose++;  break;         // 負け
         case 2: win++;   break;         // 勝ち
        }
    }

    string getName(void)          // 名前を返す
    {
        return (name);
    }

    void put_record(void)         // 勝敗数を表示
    {
```

```
        cout << name << "：" << win << "勝" << lose << "負"
                             << draw << "分けです。¥n";
    }
};


char hd[][7] = {"グー", "チョキ", "パー"};

/*--- 『じゃんけんポン』を表示 ---*/
void put_jyanken_message(void)
{
    cout << "¥n¥aじゃんけんポン … ";
    for (int i = 0; i < 3; i++)
        cout << "(" << i << ")" << hd[i] << " ";
    cout << "：";
}

/*--- 判定結果を表示 ---*/
void disp_result(int result)
{
    switch (result) {
     case 0: cout << "引き分けです。¥n";       break;
     case 1: cout << "あなたの負けです。¥n";   break;
     case 2: cout << "あなたの勝ちです。¥n";   break;
    }
}

/*--- 再挑戦するかを確認 ---*/
int confirm_retry(void)
{
    int  x;
    cout << "もう一度しますか … (0)いいえ (1)はい：";
    cin >> x;

    ret   (x);
}


int main(void)
{
    int  retry;                     // もう一度？

    Player user(0, "柴田");         // 人間
    Player comp(1, "Comp");         // コンピュータ


    time_t t;
```

ここまでがクラスの宣言です。

ここまでは、いわゆる普通の関数の定義です（クラスとは無関係です）。

```
    srand(time(&t) % RAND_MAX);      // 乱数の種を初期化

    cout << "じゃんけんゲーム開始。¥n";

    do {

        int  chand = comp.newHand();
        put_jyanken_message();       // 『じゃんけんポン』を表示
        int  uhand = user.newHand();

        cout << user.getName() << "は"   << hd[uhand] << "で、"
             << comp.getName() << "は"   << hd[chand] << "です。¥n";

        int  ujudge = (uhand - chand + 3) % 3;  // 人間の勝敗
        int  cjudge = (chand - uhand + 3) % 3;  // コンピュータの勝敗

        disp_result(ujudge);         // 判定結果を表示

        user.recordCount(ujudge);    // 勝敗回数更新
        comp.recordCount(cjudge);    // 勝敗回数更新

        retry = confirm_retry();     // 再挑戦するかを確認
    } while (retry == 1);

    user.put_record();               // 人間の結果
    comp.put_record();               // コンピュータの結果

    return (0);
}
```

■メンバ関数 **int newHand(void)**

新しい手を生成します。

プレーヤーが人間であれば、標準入力ストリーム cin（キーボード）からの読込みを行います。

プレーヤーがコンピュータであれば、0〜2 の乱数を生成します。

生成した手は、データメンバ hand に格納すると同時に、その値を返します。

■メンバ関数 **void recordCount(int result)**

勝／負／引分けの判定結果 result をもとに、勝敗回数を更新します。

■メンバ関数 **string getName(void)**

プレーヤーの名前を表す文字列を返します。

■メンバ関数 **void put_record(void)**

勝敗数を表示します。

■□■ メンバ関数の呼出し■□■

手を生成する箇所に着目しましょう。

```
int  chand = comp.newHand();

int  uhand = user.newHand();
```

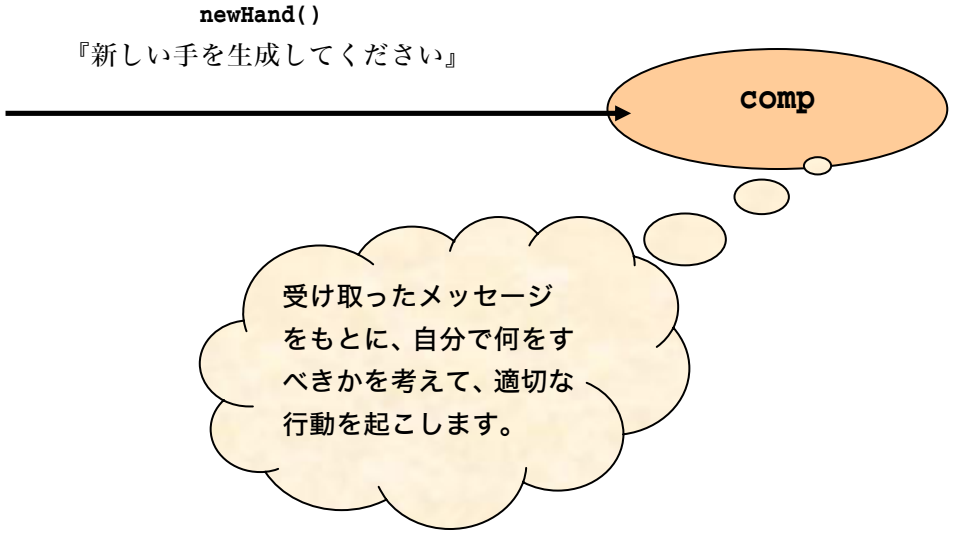
メンバ関数を呼び出す **comp.newHand()**は、**comp** オブジェクトに対して、『新しい手を生成しなさい。』という依頼を行っています。

このように、メンバ関数は、

　　オブジェクト名.メンバ関数名(引数)

という形式で呼び出します。

オブジェクト指向の世界では、**『オブジェクトにメッセージを送る』**と言います。オブジェクト **comp** に対して、『新しい手を生成してください。』というメッセージを送り、そのメッセージを受け取った **comp** は、能動的に行動を起こします（この場合は、0〜2 の乱数を生成して、その値をデータメンバ **hand** に設定するとともに、その値を返します)。



“プレーヤー”だけでなはく、“じゃんけんゲーム”もクラスとして実現すべきです。そのように変更したプログラムを示します。

```
/*
    クラスを用いたじゃんけんプログラム（Ver.2）
*/

#include  <ctime>
#include  <string>
#include  <cstdlib>
#include  <iostream>

using namespace std;

//--- プレーヤークラス ---//
class Player {

    friend class Jyanken;

    int    type;      // 0…人間／1…コンピュータ
    string name;      // 名前
    int    hand;      // 手
    int    result;    // 判定結果
    int    win;       // 勝ち数
    int    lose;      // 負け数
    int    draw;      // 引分け

public:
    Player(int t, string n)          // コンストラクタ
    {
        type = t;
        name = n;
        win = lose = draw = 0;
    }

    void newHand(void)               // 次の手
    {
        if (type == 0)               // ●人間
            cin >> hand;             //     キーボードから読み込む
        else                         // ●コンピュータ
            hand = rand() % 3;       //     乱数発生
    }

    void recordCount(void)           // 勝敗回数を更新
    {
        switch (result) {
         case 0: draw++;  break;     // 引分け
         case 1: lose++;  break;     // 負け
         case 2: win++;     break;   // 勝ち
        }
    }

    string getName(void)         // 名前を返す
    {
        return (name);
    }
```

```
    void put_record(void)        // 勝敗数を表示
    {
        cout << name << "：" << win << "勝" << lose << "負"
                             << draw << "分けです。¥n";
    }
};


//--- じゃんけんクラス ---//
class Jyanken {

    static string  hd[3];

public:
    Jyanken(void)                        // コンストラクタ
    {
        time_t t;
        srand(time(&t) % RAND_MAX);      // 乱数の種を初期化
    }

    void put_start_message(void)         // 開始メッセージを表示
    {
        cout << "じゃんけんゲーム開始。¥n";
    }

    void put_jyanken_message(void)          //  『じゃんけんポン』を表示
    {
        cout << "¥n¥aじゃんけんポン … ";
        for (int i = 0; i < 3; i++)
            cout << "(" << i << ")" << hd[i] << " ";
        cout << "：";
    }

    void put_hands(const Player &p1, const Player &p2)// プレーヤーの手を表示
    {
        cout << p1.name << "は"  << hd[p1.hand] << "で、"
             << p2.name << "は"  << hd[p2.hand] << "です。¥n";
    }

    void judge(Player &p1, Player &p2)                // 判定
    {
        p1.result = (p1.hand - p2.hand + 3) % 3;      // p1の勝敗
        p2.result = (p2.hand - p1.hand + 3) % 3;      // p2の勝敗
    }

    void disp_result(const Player &p1)                // 判定結果表示
    {
        switch (p1.result) {
         case 0: cout << "引き分けです。¥n";          break;
         case 1: cout << p1.name << "の負けです。¥n"; break;
         case 2: cout << p1.name << "の勝ちです。¥n"; break;
        }
```

```
    }

    int retry(void)                                    // 再挑戦するかを確認
    {
        int  x;
        cout << "もう一度しますか … (0)いいえ (1)はい：";
        cin >> x;

        return (x);
    }
};

string Jyanken::hd[] = {"グー", "チョキ", "パー"};


int main(void)
{
    Jyanken game;
    Player user(0, "柴田");          // 人間
    Player comp(1, "Comp");          // コンピュータ

    game.put_start_message();        // 開始メッセージを表示

    do {
        comp.newHand();                 // コンピュータの手を生成
        game.put_jyanken_message();     // 『じゃんけんポン』を表示
        user.newHand();                 // ユーザの手を生成（読み込む）

        game.put_hands(user, comp);     // 手を表示
        game.judge(user, comp);         // 判定

        game.disp_result(user);         // 判定結果を表示

        user.recordCount();             // 勝敗回数更新
        comp.recordCount();             // 勝敗回数更新

    } while (game.retry());

    user.put_record();                  // 人間の結果を表示
    comp.put_record();                  // コンピュータの結果を表示

    return (0);
}
```

■□■ クラス **Player** の変更点 ■□■

□クラス **Player** の冒頭で

```
friend class Jyanken;
```

と宣言されています。この宣言によって、クラス **Jyanken** は、クラス **Player** の《お友達》となり、その私的部にまで自由にアクセスできるようになります。

このように、クラス **Player** とクラス **Jyanken** のように、密接に関連するクラスにのみ **friend** 機能を使います。不用意に多用してはいけません。

※　この宣言は、クラス **Jyanekn** の**全**メンバ関数に対して、《私のクラスの私的部に対するアクセスを許可しますよ。》という宣言です。メンバ関数**ごとに friend** 宣言をすることもできます。

□判定結果を表すデータメンバ **result** を追加しています。

□メンバ関数 **newHand** は、値を返さないように変更しています。

□メンバ関数 **recordCount** は、引数を受け取らないように変更しています。

※判定結果を表すデータメンバ **result** を参照すればよいので。

■□■ クラス **Jyanken** について ■□■

クラス **Jyanken** を新たに追加しました。

□静的データメンバ **hd**

グー、チョキ、パーの文字列を格納する配列です。Ｃスタイルの文字列ではなく、C++ スタイルの文字列としました。

なお、たとえじゃんけんクラスのオブジェクトが複数作られても、この文字列は一つのみ存在すればよいので、st**atic** を与えて静的にしています。

□メンバ関数 **put_hads**，**judge**，**disp_result**

それぞれ、二人のプレーヤーの手を表示する関数、勝敗の判定を行う関数、判定結果を表示する関数です。クラス **Player** の **friend** ですから、私的メンバである **name**，**hand**，**result** を直接アクセスできることに注意しましょう。

コンピュータのプレーヤは、グー、チョキ、パーのいずれかを乱数として発生させますが、グーとチョキの手のみを出す、ちょっと変わったコンピュータもいるかもしれません。

プレーヤークラスを抽象クラスとして定義してみましょう。

```
/*
    クラスを用いたじゃんけんプログラム（Ver.3）
*/

#include  <ctime>
#include  <string>
#include  <cstdlib>
#include  <iostream>

using namespace std;

//--- プレーヤークラス（抽象クラス） ---//
class Player {

    friend class Jyanken;

    string name;      // 名前
    int    result;    // 判定結果
    int    win;       // 勝ち数
    int    lose;      // 負け数
    int    draw;      // 引分け

protected:
    int    hand;      // 手

public:
    Player(string n)             // コンストラクタ
    {
        name = n;
        win = lose = draw = 0;
    }

    virtual void newHand(void) = 0; // 次の手（純粋仮想関数）

    void recordCount(void)           // 勝敗回数を更新
    {
        switch (result) {
         case 0: draw++;  break;     // 引分け
         case 1: lose++;  break;     // 負け
         case 2: win++;     break;       // 勝ち
        }
    }

    string getName(void)         // 名前を返す
    {
        return (name);
    }
```

```
    void put_record(void)        // 勝敗数を表示
    {
        cout << name << "：" << win << "勝" << lose << "負"
                        << draw << "分けです。¥n";
    }
};

//--- 人間プレーヤークラス ---//
class HumanPlayer : public Player {

public:
    HumanPlayer(string n) : Player(n) { }

    void newHand(void)               // 次の手
    {
        cin >> hand;              //  キーボードから読み込む
    }

};

//--- コンピュータプレーヤークラス ---//
class CompPlayer : public Player {

public:
    CompPlayer(string n) : Player(n) { }

    void newHand(void)               // 次の手
    {
        hand = rand() % 3;           //  乱数発生（0〜2）
    }

};

//--- コンピュータプレーヤークラス（パーを出さない） ---//
class CompXPlayer : public Player {

public:
    CompXPlayer(string n) : Player(n) { }

    void newHand(void)               // 次の手
    {
        hand = rand() % 2;           //  乱数発生（0〜1）
    }

};

//--- じゃんけんクラス ---//
class Jyanken {

    static string  hd[3];

public:
```

```
    Jyanken(void)                          // コンストラクタ
    {
        time_t t;
        srand(time(&t) % RAND_MAX);        // 乱数の種を初期化
    }

    void put_start_message(void)           // 開始メッセージを表示
    {
        cout << "じゃんけんゲーム開始。¥n";
    }

    void put_jyanken_message(void)          // 『じゃんけんポン』を表示
    {
        cout << "¥n¥aじゃんけんポン … ";
        for (int i = 0; i < 3; i++)
            cout << "(" << i << ")" << hd[i] << " ";
        cout << "：";
    }

    void put_hands(const Player &p1, const Player &p2)// プレーヤーの手を表示
    {
        cout << p1.name << "は"  << hd[p1.hand] << "で、"
             << p2.name << "は"  << hd[p2.hand] << "です。¥n";
    }

    void judge(Player &p1, Player &p2)                 // 判定
    {
        p1.result = (p1.hand - p2.hand + 3) % 3;       // p1の勝敗
        p2.result = (p2.hand - p1.hand + 3) % 3;       // p2の勝敗
    }

    void disp_result(const Player &p1)                 // 判定結果表示
    {
        switch (p1.result) {
         case 0: cout << "引き分けです。¥n";           break;
         case 1: cout << p1.name << "の負けです。¥n"; break;
         case 2: cout << p1.name << "の勝ちです。¥n"; break;
        }
    }

    int retry(void)                                    // 再挑戦するかを確認
    {
        int  x;
        cout << "もう一度しますか … (0)いいえ (1)はい：";
        cin >> x;

        return (x);
    }
};

string Jyanken::hd[] = {"グー", "チョキ", "パー"};
```

```
int main(void)
{
    Jyanken game;
    HumanPlayer   user("柴田");          // 人間
    CompXPlayer   comp("Comp");          // コンピュータ

    game.put_start_message();        // 開始メッセージを表示

    do {
        comp.newHand();                  // コンピュータの手を生成
        game.put_jyanken_message();      // 『じゃんけんポン』を表示
        user.newHand();                  // ユーザの手を生成（読み込む）

        game.put_hands(user, comp);      // 手を表示
        game.judge(user, comp);          // 判定

        game.disp_result(user);          // 判定結果を表示

        user.recordCount();              // 勝敗回数更新
        comp.recordCount();              // 勝敗回数更新

    } while (game.retry());

    user.put_record();                   // 人間の結果を表示
    comp.put_record();                   // コンピュータの結果を表示

    return (0);
}
```

C ++では、純粋仮想関数をメンバとしてもつクラスは、抽象クラスとなります。
今期はここまで（本当は、じゃんけんクラスもいじくらないといけないのですが)。

参考文献

1) 柴田望洋『プログラミング講義 C++』，ソフトバンク，1996
2) B.Stroustrup（長尾高弘ら訳)『プログラミング言語 C++第 3 版』，アスキー，1998