Ｃ＋＋とＣ言語復習まとめ

目次

（１）コメント　２ページ
（２）定数　２ページ
（３）リファレンス　３ページ
（４）記憶領域の自由割り当て　４ページ
（５）変数の宣言　５ページ
（６）関数　５ページ
（７）クラス　６ページ
（８）コンストラクタ、デストラクタ　８ページ
（９）派生クラス　９ページ

Visual Ｃでは、ファイル拡張子が　『.c』なら**Ｃ言語**、『.cpp』なら**Ｃ＋＋**として処理される、

（1）コメント

Ｃ言語では/* から始まって */で終わるまでの範囲がコンパイルの対象外になる。
Ｃ＋＋でも同じコメントが使えるが、それ以外に//が使える。これは記述される位置より右の改行までがコメント範囲になる。　　改行までの範囲がコメントになる

```
int  cnt;      // 入力カウント用変数
```

（2）定数

数値などの決まった値をそのまま表す次のようなプログラムは読みにくい。

```
var = 1;

while(var == 1){
        var = 0;
```

値だけでは、意味がわかりにくい。

そこで**識別子**に値を割り付けて、その識別子を使うようにするとよい。その方法として、マクロ（#define）を使ったり列挙（enum）を使ったりしてきた。次のプログラムは FALSE, TRUE と言う文字列を使うことで、不成立、成立のデータと言うことを分かりやすくしている。

マクロ（#define）を使う場合

```
#define FALSE 0
#define TURE 1

int var;
var = TURE ;

while(var == TURE ){
        var = FALSE ;
```

#define はコンパイルする前に識別子を値に置き換える。この例では、#define で、FALSE を 0 へ、TURE を 1 へ置き換える。そして、こののち換えたファイルがコンパイル対象になる。
これは前処理機能といわれる。

列挙（enum）を使う場合

```
enum Boolean{ FALSE ,TRUE };

enum Boolean var;
var = TURE ;

while(var == TRUE){
        var = FALSE ;
```

上記例では、**enum Boolean** と言う型を作って、この型で列挙されたデータ（FALSE ,TRUE）を記憶するための変数 var を用意し、プログラミングしている。この場合、実際の内部的な値はなんでもよく、列挙される識別子の並び順で自動的に異なる値になることを利用している。識別子に割り振られる内部的な値は、指定なしの場合０からの連番で、この例なら、FALSE が 0、TRUE が 1 になる。

以上の規則は、C 言語も C++も同じです

識別子とは、プログラマが付けられる名前(文字並び)の意味で、変数や定数や構造体タグの名前などのこと！

C言語と、C＋＋では、次のようにconstの意味が微妙に違う。

```
const  int MaxData  =  500;                    （C言語では#define を使っていた。）
void main()
{
     int data[MaxData];
      //以下省略
```

C言語のconstは、変更できない変数と言う位置付けなので、エラーになるが、C＋＋では可能になる。（真の定数と使える）

なお、次のようなポインタ宣言と連係したconstの取り扱いはC言語と同じである。

●関数の中でポインタ変数p自体の内容を変更できなくなくするめの宣言

```
void func (int * const p)
{
   *p = 1;//可能
   p++;   //でエラー       以下省略
```

●関数の中でポインタ変数が**指し示す内容**を変更しないための宣言

```
void func (const int *p)
{
   p++;   //可能
   *p = 1;// でエラー        以下省略
```

const int * **const** p;
にすると、p++、*pの変更ともにできなくなるで、使うだけになる。

（3）変数の宣言

C言語の変数宣言位置は、ブロック（｛と｝で囲まれる範囲）の先頭で実行文が始まる前にまとめて宣言しなければならず、実行が始まってからの変数宣言はエラーになる。

```
{
    int  k,d = 123;
    char   c[10];
    d = 256;
    int  n;
    for(n = 0; n < 5; n++){
        d += n;
    }
}
```

ここまでが宣言

これが実行文、よってこれ以降で宣言するとエラー

対してC＋＋での変数宣言位置は、実行文が始まってからでも、変数が必要になる位置で宣言できる。

```
{
    int  k,d = 123;
    char   c[10];
    d = 256;
    int  n;
    for(n = 0; n < 5; n++){
        d += n;
    }
}
```

ここまでが宣言

実行文

宣言しても、エラーにならない。

有効範囲

C++では、forの中でも宣言でき、左と同じになる

```
for(int  n = 0; n < 5; n++){
     d += n;
}
```

なお変数が存在するとされる変数の使える範囲は、宣言位置から宣言があるブロックの最後まで、●……● で示す範囲になる。　（有効範囲：scopeと呼ぶ）

（４）リファレンス型

Ｃ言語は、引数で指定した変数を、関数の実行側で変更することはできなかった。

```
func(変数)　　/* この実行で変数を変更するプログラムは書けない */
```

変更するためには、次のように変数のポインタを渡して、関数定義側でそのポインタを介して変更する。（変更の可能性をプログラムから読み取れる長所と言える）

```
func(&変数)　　/* 変数でなく、変数のポインタを渡す */
```

この関数の定義は次のようになる。

```
void　func（型　＊ 仮引数名）
 {
　　＊仮引数名 ＝ 変更する値；
 }
```

対して、Ｃ＋＋では、次のリファレンス型を導入することで可能にしている。

```
型名 ＆ リファレンス変数ｂ ＝ 変数ａ
```

リファレンス変数ｂが変数ａで初期化されて用意される。
以降のリファレンス変数ｂは、変数ａを意味する。
このようにリファレンス変数は、必ず初期化して使う。

《具体例》

```
int  a = 5;
int& b = a; //a を参照する変数ｂを用意
b += 5;  //a を 10 に変更している。
printf("%d",a); // 10 が出力（printf("%d",b);）と記述しても同じ。
```

以上のリファレンス型も、ポインタと同じように関数との情報受け渡しでよく使われる。

```
void　func（型　＆ 仮引数名）
 {
　　仮引数名 ＝ 変更する値；
 }
```

これを利用することで、func(**変数**)の実行で、引数の**変数**を変更する関数も定義できる。しかし、プログラムの読み易さを損なうことになるので、引数の変更のために使うのは好ましくない。おもに大きなデータの、情報受け渡しにおけるオーバーヘッドを少なくするために用いる。

そして、呼び出し側の変数を変更せずに、引数を参照するために関数定義は const で修飾することを奨める。そうすることで関数の中で引数が変更できないことになるので、間違って変更する定義を記述した場合にコンパイラが指摘してくれる。またこの文法を知っていれば、そのプロトタイプ宣言から、引数を変更しない機能の関数を一目で見分けられる。

《例》

```
void func1(const POINT &pt)　　//pt の内容を参照して処理する関数
{
　　pt.x = 123;　　//pt を変更する記述はエラー！
　　省略
}
```

（５）関数

C言語で関数を定義する時、引数が違う場合には、同じ機能の関数でも関数名を異なるネーミングで作らなければならない。次のように差の絶対値を求める関数を考えた場合、C 言語ではエラーになる。C 言語の関数呼び出しは、関数の名前のみを使って呼び出すため、どちらの関数を呼び出すか特定できないためである。

```
int  absolute(int a, int b)
{
        return a > b ? a-b: b-a;
}
double absolute(double a, double b)
{
        return a > b ? a-b: b-a;
}
```

C 言語ではエラーになるが C++では可能

対して、**C++では**引数の**数や型**をチェックして、適切な関数を呼び出すことができ、このような同じ名前の関数定義が可能になっている。このように同じ名前の関数を複数定義することを**多重定義（オーバーロード）**と呼ぶ。

上記のように多重定義されている C++では、Absolute の引数の型により自動的に選択されて戻り値の型が変わる。

```
        printf("%d¥n",Absolute(5, 2));     // Absolute の戻り値が、int 型になる。
        printf("%f¥n",Absolute(5.0, 2.0));   // Absolute の戻り値が、double 型になる。
```

しかし、Absolute(5,2.0);のように、一方が double 型でもう一方が int 型の指定をすると、コンパイラはどちらを実行するべきか判断できなく、『ｵｰﾊﾞｰﾛｰﾄﾞ関数があいまいです。』のエラーになる。このような場合は、明示的にキャストであいまいにならない記述が必要になる。

また、C++では**デフォルト引数**と呼ばれる次のような表記方法ができ、引数を省略した呼び出しも可能になっている。例えば次のように、引数 b を指定しない時に 10 の値を使わせる場合、

```
int  Absolute(int a, int b = 10)
{
        return a > b ? a-b: b-a;
}
```

このように定義される場合、x = Absolute(6, **10**); は、次のように **10** を省略して

```
  x = Absolute(6);
```

と呼び出すことができる。なお、デフォルト引数はプロトタイプ宣言でも使えるが、定義とプロトタイプ宣言の両方で指定することはできない。

その他、C++では C 言語の引数付きマクロの不具合に対応するため、インライン関数と呼ばれる関数定義も使える。関数定義の先頭で、inline と記述して定義した関数は、関数呼び出し位置に、関数の機能を実現する翻訳コードを埋め込む。これにより、マクロと同程度の高速化（関数呼び出しの時間が少ない）で、マクロの副作用も起きないプログラムが書ける。

例

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

呼びだしする記述は、通常の関数と同じ。

```
int k = 5, j = 10, m;
m = max(k++, ++j);
// m に 11 が記憶される。
```

（６）記憶領域の自由割り当て

まず、通常の変数を使ったプログラム例を下記に示す。

内容は、int 型変数、int 型配列、名前と年齢を記憶する構造体の変数と、その配列を用意し、設定（代入）するだけの確認プログラムである。

```
struct PERSON {
       char name[50]; /*名前 */
       int age;       /* 年齢 */
};
void main()
{
  int i;
  int a[10];
  struct PERSON d;
  struct PERSON D[10];

  i = 123;
  a[9] = 456;
  d.name[0] = '¥0';
  D[9].name[0] = '¥0';
}
```

各データを記憶する変数を用意

上記は、各データを記憶する変数を用意していた。こんどはこれと同じプログラムを、ポインタ変数だけにして、実際のデータ記憶域は実行時に用意し、ポインタを介したデータを管理してみる。

Ｃ言語では **malloc 関数**で用意し、不要になったら **free 関数**で開放する。（左下プログラム）、対してＣ＋＋では **new 演算子**と **delete 演算子を**使うことになる。（右下プログラム）

Ｃ言語

```
#include <stdlib.h>

struct PERSON {
       char name[50]; /*名前 */
       int age;       /* 年齢 */
};
void main()
{
  int *pi;
  int *pa;
  struct PERSON *pd;
  struct PERSON *pD;

  pi = (int*) malloc(sizeof(int));
  *pi = 123;
  pa = (int*) malloc(sizeof(int) * 10);
  pa[9] = 456;
  pd= struct PERSON*)malloc(sizeof(struct PERSON));
  pd->name[0] = '¥0';
  pD=(struct PERSON*)malloc(sizeof(struct PERSON)*10);
  pD[9].name[0] = '¥0';

  free(pi);
  free(pa);
  free(pd);
  free(pD);
}
```

各データを管理変数を用意
（データを記憶する変数はない！）

記憶域を用意

C++

```
#include <stdlib.h>

struct PERSON {
 char name[50]; //名前
  int age;     // 年齢
};
void main()
{
  int *pi;
  int *pa;
  PERSON *pd;
  PERSON *pD;

  pi = new int;
  *pi = 123;
  pa = new int [10];
  pa[9] = 456;
  pd = new PERSON;
  pd->name[0] = '¥0';
  pD = new PERSON[10];
  pD[9].name[0] = '¥0';

  delete pi;
  delete []pa;
  delete pd;
  delete [] pD;
}
```

まず最初の C 言語との違いが 11,12 行に存在する。これは、C++で構造体変数を宣言する時に **struct** の記述が省略できることを意味する。

さてC言語プログラムは、記憶したい情報の発生したタイミングで、その情報を記憶するに必要なサイズのメモリを **malloc** 関数で取得し、その戻り値をポインタ変数で管理させるプログラミングテクニックをよく使う。**malloc** 関数で得られたポインタが管理する記憶空間は、その内容が不要になった時点で､次に再び記憶域が必要になった時に再利用できるように **free** 関数で開放する。即ち､必要なタイミングで変数の変わりに使えるメモリを用意し､不要になったら開放することで、メモリを効率的に使う訳である。このようなメモリ操作は、『**動的なメモリ割り当てと開放**』と呼ばれる。そして C++のオブジェクト指向プログラミングではこのテクニックをもっと頻繁に使うことになる。C++では **malloc** 関数の代わりに new **演算子**を使えて、**free** 関数の変わりに delete **演算子**を使える。C 言語と C++を比較するため、リストでは同一行で並べている。

14 行で用意した記憶域へ 15 行目で設定し、
16 行で用意した記憶域へ 17 行目で設定し、
18 行で用意した記憶域へ 19 行目で設定し、
20 行で用意した記憶域へ 21 行目で設定している。

malloc 関数、new **演算子**両者ともにポインタ変数の有効範囲が外れても確保されたメモリ領域は自動的に開放しない。この記憶域を管理しているポインタ変数がなくなると、再利用できない記憶領域となる。（このようにプログラムで管理できない領域ができることを**メモリーリーク**と呼ぶ）
この管理できなくなった記憶域は、プログラムが終了する時点でシステムが開放してくれる。しかし連続運転するプログラムなどでは適切な所でメモリ開放をしないと管理できない記憶域が少しずつ蓄積されようなバグができる。このようなバグ（開放し忘れ個所があるかもしれないバグ）の発見は困難で、日頃からプログラマの責任として開放すべき所で開放することを奨める。リスト上での使用済み領域開放は、23～26 行で行っている。

new は、**malloc** 関数に比べてポインタのキャスト変換や、サイズを計算する必要が無くなるので使いやすいものになっている。以下に一般的な new **演算子**と delete **演算子**の構文を示す。

```
ポインタ変数 = new 型;    //指定型のオブジェクトをメモリに割り当てる。
delete ポインタ変数;      //ポインタ変数が指す記憶域を開放する。
```

C 言語のテクニックでよく使う複数並べて配列のように使う記憶域は、次のように指定する。

```
ポインタ変数 = new 型[個数];    //指定型のオブジェクトを指定個数分用意する。
delete []ポインタ変数;          //ポインタ変数が指す記憶域を開放する。
```

このように new 配列で生成したポインタに対して delete を使う場合、[ ]の指定が前方にないと、全体の記憶管理に不整合が起き、以降正しく使えなくなるので注意しなければならない。

なお、**malloc** 関数、**free** 関数に対応する new **演算子**、delete **演算子**は、類似する機能になっているが、異なる記憶空間で管理される。（前者のメモリ空間はヒープ領域と呼ばれ、後者はフリーストアまたは自由領域と呼ばれる）当然に new **演算子**で取得したポインタに対して **free** 関数は使えないので混同しないようにしなければならない。（C++では new **演算子**、delete **演算子**だけで **malloc**、**free** を使わなければよい）**malloc**、**free** との最も大きな違いは、new **演算子**を使った場合に**コンストラクタ**が実行され、delete **演算子**使った場合にデストラクタが実行されてから開放することである。（コンストラクタとデストラクタの説明は後述）

（7）クラス

クラスとはプログラムする時に、必要に応じてまとめた型と言える。つまり**C 言語の構造体と同じで、このままクラスへと拡張している。よってこれ以降を、構造体と呼ばずにクラスと表現する。そしてクラスで定義される変数やその記憶空間を『オブジェクト』と呼ぶ。**

まとめた理由は、そのまとまったデータ(オブジェクト)に対して固有の操作をする場合が多いからと言える。しかし C 言語での構造体に対する関数は、グローバルな関数として定義していたので、プログラムリストを見た時にその構造体に対する専用の関数か否かがすぐ分からなかった。

C++では、関数をクラス（構造体）固有の機能として定義できる。その記述方法は次のようにクラス定義中で、データメンバと同じように宣言すればよい。これは**メンバ関数**や**メソッド**と呼ばれる。

つまりクラスは、オブジェクトの設計書で、構造体のデータメンバ以外に、操作や振る舞いなどの機能を、関数プロトタイプ宣言のイメージで宣言し、定義する形態になる。そして、クラスで宣言した変数や **new** で生成されるものがオブジェクト（インスタンスと呼ばれることもある）になる。

以下の名前と年齢を管理するプログラムで、C の構造体と C++のクラス定義を比較する。

C 言語

```
struct PERSON {
        char name[64]; /*名前メンバ*/
        int  age;      /*年齢メンバ*/
};

void init(struct PERSON *dp,char *s,int d)
{
  /*引数の情報を dp が指し示す領域に設定する */
}

void disp(const struct PERSON *dp)
{
  /*dp が指し示す内容を表示する */
}

int input(struct PERSON *dp)
{
  /*キーから入力したデータを、 dp が指し示す
    領域に設定する（正常入力で１を返す）  */
}
```

C++

```
struct PERSON {
  char name[64]; //名前データメンバ
  int  age;      //年齢データメンバ
  void init(char *s, int d);
  void disp()const;
  int input();
};

void PERSON::init(char *s, int d)
{
 //引数の情報を自身の name と age に設定
}                (this)  this->name
                          や this->age と表現

void PERSON::disp()const
{
 //自身の内容を表示する
}  (this)

int PERSON::input()
{
  //キーから入力したデータを、
  //自身に設定する（正常入力で１を返す）
}      (this)
```

メンバ関数の宣言

以上が PERSON 型に対する機能定義。　　以下で簡単に使ってみる

```
void main()
{
    struct PERSON D1,D2;

    init(&D1,"Tarou",18);

    input(&D2);

    disp(&D1);
    disp(&D2);
}
```

```
void main()

    PERSON D1,D2;

    D1.init("Tarou",18);

    D2.input();

    D1.disp();
    D2.disp();
}
```

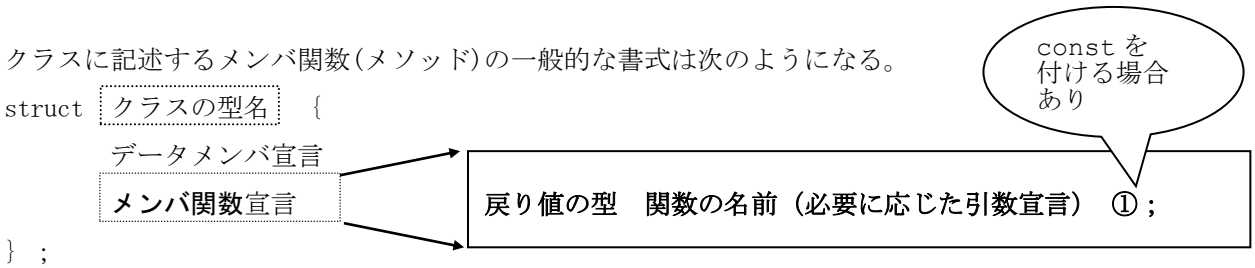
オブジェクト変数の宣言

オブジェクトへの設定

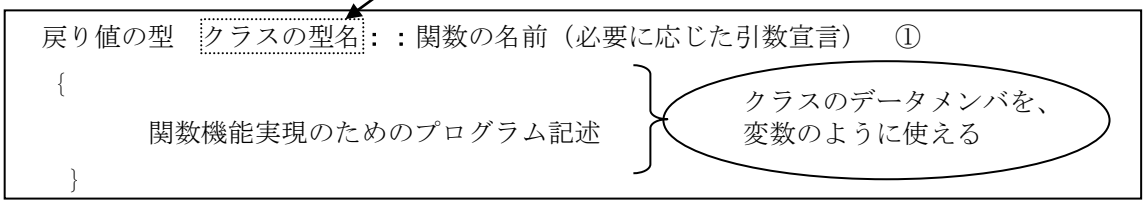
オブジェクトへキー入力

オブジェクトの表示

クラスに記述するメンバ関数(メソッド)の一般的な書式は次のようになる。



・**メンバ関数**（メソッド）宣言は、プロトタイプ宣言と同じような役割になる。
・**メンバ関数**も多重定義やデフォルト引数が使える。(関数定義ではデフォルト引数を与えない..)
・データメンバ宣言と**メンバ関数**（メソッド）宣言の**数や、順番**は任意である。
さてメンバ関数（メソッド）上記**宣言**に対して、次のようなメンバ関数（メソッド）の**定義**が必要になり、ここに実態である機能をプログラミングする。
方法は、C 言語の関数定義に対して、クラスの型名の指定がコロン２つと共に追加されるだけのイメージである。

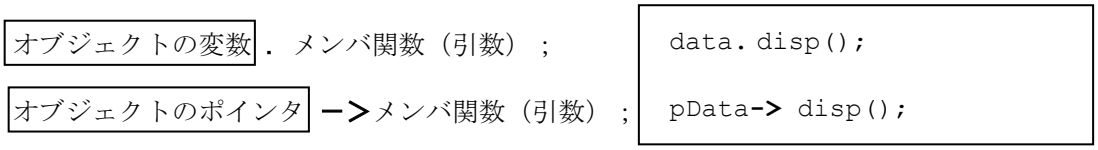


この中では**メンバ名を表現するだけ**で呼び出したオブジェクト**自身**のデータメンバにアクセスできる。しかし、メンバ関数（メソッド）内部で宣言したローカル変数と区別するため、**this** を使ってこの『オブジェクトのメンバ』と言うことを明確にした方が分かりやすい。
this は、メソッドを呼び出したオブジェクトの**ポインタ**を表現する予約語になっている。

```
void PERSON::disp()    ①
{
        printf("名前：%s 年齢：%d¥n",  this->name,    this->age);
}
```

なお、C と C++比較リスト（前ページ）では、disp メソッド宣言に **const** の記述がある（上記説明では①の個所）。この const が付くメンバ関数（メソッド）は、C との比較から分かるように、メソッド呼び出し元のオブジェクト変更を禁止するもので、結果的にオブジェクトを保護していることになる。これによりクラス利用者は、クラス定義部分（struct { ～ } ）を見て、呼び出し側のオブジェクトを変更していないメソッドかどうか判断して使うことができる。

メンバ関数の呼び出しは、構造体メンバと同じで、そのオブジェクトに対してドット『. 』で指定する。またオブジェクトへのポインタを表現する記述に対しては『->』で指定する。



メンバ関数（メソッド）は、クラス定義内の宣言位置でプログラムすることも可能で、これは inline を記述しなくても自動的に**インライン関数**になる。

インライン関数:関数呼び出しタイプでなく、コードが埋め込まれるタイプの翻訳になる。

```
struct PERSON {
    char name[64]; //名前データメンバ
    int  age;      //年齢データメンバ
    void init(char *s, int d); //設定メンバ関数
    void disp()  { printf("名前：%s 年齢：%d¥n",name,    age);}
    int input();  //入力メンバ半数
};
```

（8）コンストラクタ、デストラクタ

変数は一般に、何らかの初期値を必要としたり、他の変数と連係した設定処理を必要とする場合が多い。そして C++では、オブジェクトの記憶域が生成される時（ポインタ以外の変数も）、自動的に実行する関数を、クラス作成者が提供できる仕組みになっている。これは**コンストラクタ**と呼ばれ、**クラス名と同じ名前**の特別なメンバ関数で、戻り値の型を記述しない形態に定義する。

また、逆に変数を破棄する時に自動的に実行させるメソッドは、**デストラクタ**と呼ばれる。

オブジェクトの破棄に伴う後始末の機能もクラス設計者が定義できる訳だ。これは、～『チルダ』を付けたクラス名と同じ名前にして**引数なし**で戻り値の型を記述しない形態に定義する。次の例で Integer クラスのコンストラクタ、デストラクタの動きを確認しよう。

```
01  #include <stdio.h>
02  struct Integer {                //Integer クラスの定義
03          int t;
04
05          Integer(int );          //引数付コンストラクタ宣言
06          ~Integer();             //デストラクタ宣言
07  };
08  Integer::Integer(int d)         //変換コンストラクタ定義
09  {
10          t = d;
11          printf("t が%d の記憶域を用意しました¥n",d);
12  }
13  Integer::~Integer()     //デストラクタ定義
14  {
15          printf("%d の記憶域を破棄しました¥n",this->t);
16  }
17
18  void main()
19  {
20          ①
21          Integer v1(2);
22          Integer *pv;
23          {
24                  Integer v2(5);
25
26          }
27                  ③
28  }
```

②オブジェクト生成でないので
コンストラクタは実行されない。

変数 v2 の存在範囲

実行結果

```
t が 2 の記憶域を用意しました
t が 5 の記憶域を用意しました
5 の記憶域を破棄しました
2 の記憶域を破棄しました
Press any key to continue
```

実行結果より②のポインタの変数の用意では、オブジェクトが生成されないのでコンストラクタが実行されていないことが分かる。コンストラクタはオブジェクトの記憶域が用意される時だけ実行し、記憶域の開放する③でデストラクタが実行している。この例では引数付コンストラクタは存在するが、引数付のないコンストラクタは存在しない。引数付コンストラクタは、引数の型からの**変換コンストラクタ**と呼ばれ、引数がないコンストラクタは**デフォルトコンストラクタ**と呼ばれる。デストラクタは引数がなしなので一つしか作れないが、コンストラクタは引数の数や種類を変えることで複数作れる。もし全くコンストラクタを定義しない場合は、コンパイラが生成するデフォルトのものが使われる。しかし注意しなければならないのが上記例のように**デフォルトコンストラクタ**を定義せずに**変換コンストラクタ**を定義した場合である。この時はコンパイラがデフォルトコンストラクタを用意してくれない。よって、① で **Integer** v と用意すると、『デフォルトコンストラクタがありません』のコンパイルエラーになってしまう。

これは配列を使う場合も同じで、仮に　①　で **Integer** *pv = new Integer[5];と記述する場合、5 個のオブジェクトが生成ごとに**デフォルトコンストラクタ**を呼び出すことになり、デフォルトコンストラクタがないのでエラーになる。

以下に**デフォルトコンストラクタ**も含めた幾つかのコンストラクタ実装例を示す。

```
#include <stdio.h>
#include <stdlib.h>
struct Integer {
    int i;
    Integer();                  //デフォルトコンストラクタ
    Integer(const int );        //引数付コンストラクタ（変換コンストラクタ）宣言
    Integer(const char *);      //引数付コンストラクタ（変換コンストラクタ）宣言
    Integer(const Integer &);//コピーコンストラクタ
    ~Integer();         //デストラクタ宣言（一つしか作れない）
};
Integer::Integer()    //デフォルトコンストラクタ
{
    i = 0;
}
Integer::Integer(const int d)  //int 型引数付きコンストラクタ定義
{
    i = d;
}
Integer::Integer(const char *s)    //char *型引数付きコンストラクタ定義
{
    i = atoi(s);//文字列を数値に変更して初期化
}
Integer::Integer(const Integer &d) //コンストラクタ定義
{
    i = d.i;
}
Integer::~Integer()       //デストラクタ定義
{
    printf("%d の記憶域を破棄しました¥n",this->i);
}
Integer func(Integer arg = 0) //コンストラクタでないグローバルな関数
{
    return 3;  //戻り値で、変換コンストラクタにより、Integer が生成
}
void main()
{
    Integer v1;             //①デフォルトコンストラクタが実行
    Integer *p = new Integer[3];   //②デフォルトコンストラクタが 3 回実行
    Integer v2 = 2;     //③これは、Integer v2(2);  と記述したのと同じになる。
    Integer v3 = "3";   //④これは、Integer v3("3");  と記述したのと同じになる。
    Integer v4 = v3;   //⑤
    Integer *p2 = new Integer(4);  //⑥
    Integer v5 = func(1);       //⑦
    Integer v6 = func(v4);   //⑧
    Integer a[] = {2,"3",v5};//⑨
}
```

変換コンストラクタを用意することで、初期化の＝操作で変換したり、new で生成する③、④のように括弧の中で指定したり、⑥のように引数受け渡し時や、return の戻り値で変換できたり、⑦のように配列の初期化で変換できるようになる。クラスの参照渡しを引数にするコンストラクタは特にコピーコンストラクタと呼ばれ、⑤の変数用意や⑧の引数生成時に使われる。このように C++ によるオブジェクトは様々な初期化する方法が存在する。その代わり、自前のコンストラクタを用意した時点で、C 言語のような構造体の初期化方法は、許されなくなっている。この例であれば、v.i に 123 が記憶される変数を用意する時の、Integer v = { 123 }; と言う初期化は許されない。仮にコンストラクタを一切作らなかった場合は、C 言語と同じ初期化が可能になる。

（9）派生クラス

クラスを定義する時、既にあるクラスのデータや機能を取り込んで定義することができる。

```
class 新規クラス名 : public 既にあるクラス名 {
        新たなメンバ宣言；
};
```

この時、新規クラスを「既にあるクラス」の**派生（derived）クラス**と呼び、既にあるクラスを「新規クラス」の**基底（base）クラス**と呼ぶ。そして、派生クラスでは、基底クラスのデータ（データメンバ）や機能（メソッド）を**継承（inherit）**することになる。例えば次のように

```
//人のクラス
struct  PERSON {
    char name[64]; //名前データメンバ
    int  age;      //年齢データメンバ

    PERSON(char *s, int d);//コンストラクタ



    void  setPerson(char  *s  ,  int  d);



    void disp();          //表示メソッド
};
```

person.h

```
#include <stdio.h>
#include <string.h>
#include "person.h"


//コンストラクタ定義
PERSON::PERSON(char *s, int d)
{
    this->setPerson(s,d);
}
void PERSON::setPerson(char *s, int d)
{
    strcpy(this->name,s);
    this->age = d;
}
void PERSON::disp()
{
    printf("名前:%8s /年齢%4d¥n",
            this->name, this->age);
}
```

person.cpp

名前と年齢を記憶し、設定と表示メソッドを持つ **PERSON** クラスが作られていた場合、これを継承して学生用のクラスを作成すると、次のように人の名前や年齢のデータメンバを新たに用意する必要がなくなり、学生専用に使う成績の評価データメンバ（**grade**）などを追加用意すればよい。

```
//人を継承した学生クラス
struct STUDENT : public PERSON{
    int  grade;  //成績（グレード）

    STUDENT();//コンストラクタ


    //引数つきコンストラクタ定義
    STUDENT(char *s, int d, int g);




    void disp();


    void setGrade(int g) { grade = g; }
};
```

student.h

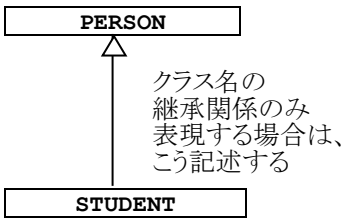
```
#include <stdio.h>
#include "person.h"
#include "student.h"
//デフォルトコンストラクタ
STUDENT::STUDENT():PERSON("",-1)
{
    this->grade = 3; //評価グレード
}
STUDENT::STUDENT(char *s, int d, int g)


:PERSON(s,d)


{
    this->grade = d;
}
void STUDENT::disp()


{
    //基底クラス
}
```

コンストラクタ初期設定子

student.cpp

以上で、**PERSON** クラスを継承した **STUDENT** クラスを定義したことになり、**STUDENT** クラスのオブジェクトには、**PERSON** クラスの名前、年齢も記憶でき、**PERSON** クラスを設定する **SetPerson** メソッドや表示機能も備わることになる。この継承関係は、UML（Unified Modeling Language）のクラス図で右記のように表す。

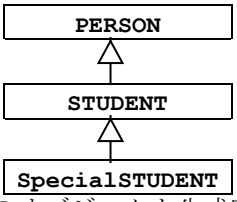
上位（**親**）クラスの **PERSON** を、**STUDENT** の**スーパクラス**と呼ぶ。



また、下位(子)クラスの **STUDENT** は、**PERSON** のクラスの**サブクラス**と呼ばれる。サブクラスからスーパクラス方向へ図のように矢印を記述して継承関係を表す。この関係は、汎化される人のクラスより特化した学生クラスが作られる継承の構造で、**「汎化・特化の関係」**と呼ばれたり、「学生 is a 人.」と表現できることから**「is a 関係」**と呼ばれたりしている。

さて、前述のようなクラスコードが用意される時、次のようなプログラムが記述でき、以下に説明する。

ここでは、新たに STUDENT を継承した SpecialSTUDENT クラスを作成している。




```
01 #include <stdio.h>
02 #include <string.h>
03 #include "person.h"
04 #include "student.h"
05 struct SpecialSTUDENT : public STUDENT {//特別学生
06     int scholarship; //奨学金
07     SpecialSTUDENT();
08     void SetStudent(STUDENT *p); //学生データを設定
09     void disp(); //表示
10 };
11 SpecialSTUDENT::SpecialSTUDENT()
12 {//コンストラクタ
13     scholarship = 50000;
14 }
15 void SpecialSTUDENT::SetStudent(STUDENT *p)
16 {//学生データを設定
17     this->setPerson(p->name,p->age);
18     this->setGrade(p->grade);
19 }
20 void SpecialSTUDENT::disp()//表示
21 {
22     STUDENT::disp();
23     printf("    奨学金:%d 円¥n",this->scholarship);
24 }
25 void main()
26 {
27     STUDENT student1("安部",19,4);
28     STUDENT *pStudent2 = new STUDENT("伊藤",19,5);
29     student1.disp();
30     student1.PERSON::disp();
31     pStudent2->disp();
32     SpecialSTUDENT *temp=new SpecialSTUDENT();//①
33     temp->SetStudent(pStudent2);
34     temp->disp();
35     delete pStudent2;
36     pStudent2 = temp;
37     pStudent2->disp();
38 }
```

実行⇒結果

```
名前:    安部 /年齢  19        29 行実行
   成績：4
名前:    安部 /年齢  19        30 行実行
名前:    伊藤 /年齢  19        31 行実行
   成績：5
名前:    伊藤 /年齢  19
   成績：5                     34 行実行
      奨学金:50000 円
名前:    伊藤 /年齢  19        37 行実行
   成績：5
```

●派生クラスのオブジェクト生成時は、自動的に親クラスのコンストラクタから順番に実行する。（デストラクタ実行順は逆になる）

即ち、27 行、28 行、32 行で全て **PERSON** のコンストラクタが実行している。32 行の動作順を追うと、SpecialSTUDENT オブジェクトは、person.cpp の 9 行目が実行し、student.cpp の 7 行目、最後に左リストの 13 行が実行されて生成される。この時、親クラスにデフォルトコンストラクタがある場合は、左 11 行のようになにも記述しないが、親クラスに変換コンストラクタしかない **STUDENT** クラスでは、student.cpp の 10 行のように：「コロン」の後で親クラスのコンストラクタを呼び出す形態になる。

（このように、親のコンストラクタを指定して呼び出すこともできる）

●上記 17、18 行に見られるように、親クラスのメソッドも呼び出せる。

●上記 29 行目や 31 行目で、親クラスに同じ名前のメソッドがあっても、その変数の型に合ったメソッド（disp）が呼ばれる。しかし左 22 行や、30 のように、親クラスのメソッドを::「コロンが 2 つ」で直接に親を指定することで呼び出すこともできる。

この::は、**スコープ解決演算子**と呼ばれる。

●36 行にあるよう、親クラスのポインタ変数へ、子クラスのポインタを直接代入できる規則がある。しかし逆はできず、その場合は次のようなキャストが必要になる。

```
temp = (SpecialSTUDENT *)pStudent2;
```

●34と37行実行を比較すると、同じオブジェクトでも、管理する変数の種類により違う結果になってしまう。オブジェクトに対応したメソッドを実行させるは、person.h の ⓥ で、**virtual** 宣言する必要がある。この詳細は後述する！

なお、以下のようキャストすれば、SpecialSTUDENT(特別学生)として無理やり表示することも可能である。

```
((SpecialSTUDENT *)pStudent2)->disp();
```

（10）仮想メソッドとポリモルフィズム

次のような PERSON クラスと STUDENT クラスを、再び使ってみる。

```
//人のクラス
struct  PERSON {
    char name[64];//名前データメンバ
    int  age;     //年齢データメンバ
    //コンストラクタ
    PERSON(char *s, int d);

    //表示メソッド
    virtual void disp();
};
```

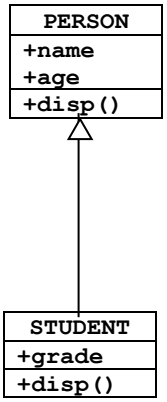
person.h

Ⓥ virtual が付くか否かが、どう変わるか確認する。

```
01 #include <stdio.h>
02 #include <string.h>
03 #include "person.h"
04
05
06 //コンストラクタ定義
07 PERSON::PERSON(char *s, int d)
08 {   strcpy(this->name,s);
09     this->age = d;
10 }
11 void PERSON::disp()
12 {
13     printf("名前:%8s /年齢%4d¥n",
14             this->name, this->age);
15 }
```

person.cpp

クラス図は
次のように書ける。




```
//人を継承した学生クラス
struct STUDENT : public PERSON{
    int  grade;  //成績（グレード）

    //引数つきコンストラクタ定義
    STUDENT(char *, int , int );



    void disp();


};
```

student.h

```
01 #include <stdio.h>
02 #include "person.h"
03 #include "student.h"
04
05
06 //引数つきコンストラクタ定義
07
08
09 STUDENT::STUDENT(char *s, int d, int
10
11 g)
12
13 :PERSON(s,d)
14
15
16 {
```

student.cpp

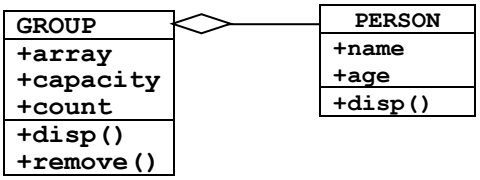
中間の仕切り内
がデータメンバ

最後の仕切りに
メソッドが並ぶ

以上クラスがある時、PERSON クラスのポインタを包含する GROUP クラスを次のように定義し、使ってみる。

```
01  #include <stdio.h>
02  #include <string.h>
03  #include "person.h"
04  #include "student.h"
05  struct GROUP {
06          PERSON **array;//array[0], array[1], と配列として使う。要素は(PERSON *)型
07          int capacity;   //上記配列の割り当て数
08          int count;      //記憶データの個数
09
10          GROUP(int n); //コンストラクタ
11          void disp();   //グループ全体の表示
12          void remove(int i);
13  };
14
15  GROUP::GROUP(int n)
16  {
17          this->array = new PERSON *[n];
18          this->count = 0;
19          this->capacity = n;
20  }
21
22  void GROUP::disp()
23  {
24          int i;
25          for(i = 0; i < this->count; i++){
26                  this->array[i]->disp();      //①各要素表示
27          }
28  }
```

GROUP クラスは、**PERSON** クラスを持っているような関係で、**「集約の関係」**や**「包含の関係」**と呼ばれる。この関係は、**GROUP** has a **PERSON** と言えるので、「has-a の関係」と呼ばれることもある。この場合のクラス構造図は、次のように菱形矢印で表現される。



```
29  void GROUP::remove(int i)
30  {
31          delete this->array[i];
32          for(i++ ; i < this->count; i++){ //詰める繰り返し
33                  this->array[i-1] = this->array[i];
34          }
35          this->count--;
36  }
37
38  void main()
39  {
40          GROUP group = 100;    //100 個の要素を管理させる。
41          group.array[group.count++] = new PERSON("鈴木",24);//データを追加設定
42          group.array[group.count++] = new STUDENT("佐藤",20,4);
43          group.array[group.count++] = new PERSON("佐々木",19);
44          group.disp();         //グループ全体の表示
45          putchar('¥n');
46          group.remove(0);      //0 の要素を削除
47          group.disp();         //グループ全体の表示
48  }
```

実行結果

```
名前:    鈴木 /年齢  24
名前:    佐藤 /年齢  20
   成績：4                    } 45 行の group.disp()
名前:  佐々木 /年齢  19

名前:    佐藤 /年齢  20
   成績：4                    } 47 行の group.disp()
名前:  佐々木 /年齢  19
```

このプログラムの実行結果は右のようになる。基底クラスで作った **virtual** 関数を、派生クラスで再び定義することを、仮想関数の**オーバーライド**(override)と呼ぶ。この例における disp メソッドがそうである。この時、基底クラスで宣言した戻りや引数の並び、型は完全に一致しなければならない。そうしないとオーバーライドではなく、オーバーロード（関数多重定義）になってしまう。またオブジェクト指向プログラミングのハイライト的機能である**ポリモーヒィズム(polymorphism：多形性、多態性、多様性)**を実現するには、基底クラスのメソッドを仮想（virtual）関数にする以外にも、オブジェクトをポインタで管理しなければならない規則がある。この例で言うところの array 要素である。40 行～43 行で、それぞれ array[0]と、array[2]には、PERSON オブジェクトのポインタ、array[1]には、STUDENT オブジェクトのポインタを管理させている（基底クラスのポインタ変数へ派生クラスのポインタが代入できたので）。そして、44 行などの group.disp()実行の中の 26 行目①で、array[i]->disp();により表示させている。これで、array[i]が管理しているオブジェクトが PERSON なら、PERSON クラスの仮想メソッドである disp()が呼び出され、array[i]が管理しているオブジェクトが STUDENT なら、STUDENT クラスの仮想メソッドである disp()が**自動的**に呼び出されている。

このように、オブジェクトが呼ぶ出すべきメソッドを、自律的に選択して呼び出してるこれが**ポリモーヒィズム**になる。C++言語ではメソッドの仮想化と、オブジェクトのポインタ管理のどちらが欠けても実現しない。また、**virtual** 宣言は基底クラスで一度宣言すると、派生クラスでオーバーライドしたメソッドは自動的に仮想メソッドになる。仮想でないメソッドや、ポインタでない変数から呼び出されるメソッドは、コンパイルリンク時に呼び出すメソッドのスタートアドレスが決定し、プログラム実行前から確定しているメソッドになることから、**静的なバインディングのメソッド**とよばれ、ポリモーヒィズムが働くメソッドと区別される。ポリモーヒィズムが働く時は、プログラム実行時のオペーレーションによって生成されるオブジェクトが変化しても対応する。

よってプログラムスタート後に、オブジェクトを識別し、オブジェクト自身によりメソッドが選べる機構が必要になる。そのために、C++では**仮想メソッドテーブル**と呼ばれる仮想メソッドのスタートポインタテーブルをクラスごとに用意し、ポリモーヒィズムを適用させるオブジェクトには、このテーブルを指し示すためのポインタメンバを、見えないメンバとして持たせることで実現している。ポインタ変数はこの**仮想メソッド**テーブルへのポインタから自身が実行すべきメソッドを決める訳である。実際 PERSON 定義に virtual 宣言がある時の sizeof(PERSON)結果は、virtual 宣言がない時より、このテーブルへのポインタ(4byte)メンバ分だけ多い。(VC のデバック時に見える仮想テーブル名は vptr になっている。)

（11）クラスのカプセル化

クラスを設計で各種メンバを用意するが、その中には利用者から**直接アクセスされては困るメンバ**を作らなくてはならない場合がある。そして C++では、利用者が間違って**直接アクセスしては困るメンバ**を使うプログラムができないようにする規則(アクセスする記述に対してコンパイルエラーを起こさせる)が、文法として盛り込まれている。それらはアクセス指定子と呼ばれ、次の3通りを記述する形態になっている。

private	非公開:このメンバが宣言されるクラスのメソッド中でのみアクセス可
protected	限定公開:このメンバが宣言されるクラスとその派生クラスのメソッド中でのみアクセス可
public	公開:任意の所でアクセス可

これらは、クラスの利用者に対してアクセス可能範囲の制限指定を意味し、それは隠して見えなくするイメージであることから**メンバの隠蔽**と呼ばれる。そしてこの考慮を施すことを**カプセル化**と呼ぶ。例として、前述の PERSON クラスを再び考えてみる。

```
struct  PERSON {
    char name[64];//名前データメンバ
    int  age;      //年齢データメンバ
    PERSON(char *s, int d); //コンストラクタ 引数で、name と age を設定する。
    virtual void disp();//表示メソッド
};
PERSON::PERSON(char *s, int d) {     //コンストラクタ定義
    strcpy(this->name,s);
    this->age = d;
}
void PERSON::disp(){

    printf("名前:%8s /年齢%4d¥n",this->name, this->age);
}
```

以上のように作られたクラスでは、次の②のように、クラスの利用の立場から、name や age が変更できる。

```
void main()
{
        PERSON *p1 = new PERSON("鈴木",24);          //①
        ‥省略･･･
        strcpy(p1->name,"佐々木");
        p1->age  = 20;
```

② ← アクセス不可のエラーにしたいなら メンバを private か protected にする

さて、ここでこの PERSON クラスの設計者は、生成後に名前や年齢の変更がない目的のクラスとして設計したいとする。(あるメンバの変更がないと言う前提で、他の連動プログラムが作られることはよくあるので･･･)
このような場合、クラスの設計者は、②のメンバを直接変更することを許さないとして、name と age メンバを次のように private 宣言すればよい。これで、上記②の個所では、『private メンバにアクセスできません。』と言う旨のコンパイルエラーが出て、利用が制限されたことになる。(カプセル化されたことになる)
なお、コンストラクタや disp メソッドは、利用できるようにしたいので、④で **public** を宣言している。
これを忘れると、上記の生成(コンストラクタ)の①でもエラーになってしまう。

なお、範囲は、次のアクセス指定まで有効になる。

```
struct  PERSON {
private:        //③
    char name[64];//名前データメンバ
    int  age;      //年齢データメンバ
public:         //④
    PERSON(char *s, int d); //コンストラクタ 引数で、name と age を設定する。
    virtual void disp();//表示メソッド
};
PERSON::PERSON(char *s, int d) {     //コンストラクタ定義
    strcpy(this->name,s);
    this->age = d;
}
void PERSON::disp(){
    printf("名前:%8s /年齢%4d¥n",this->name, this->age);
}
```

メソッドは同じクラスのデータメンバであれば、アクセス指定に関係なく、自由にをアクセスできる。

つまり前述例の PERSON は、main で利用すると、オブジェクトの生成と表示だけしかできないクラスになる。
なお、この PERSON クラスは、右のように **class** で宣言した場合と全く同じである。
つまり、先頭で private:指定をしなくても デフォルトのアクセス指定が **private** になるのが **class** で、デフォルトのアクセス指定が **public** になっているのが **struct** である。
本質的に **struct** と **class** は、このデフォルトのアクセス指定が違うだけである。

```
class  PERSON {
        char name[64];//名前データメンバ
        int  age;     //年齢データメンバ
public:
        //コンストラクタ
        PERSON(char *s, int d);

        //表示メソッド
        virtual void disp();
};
```

（このデフォルトアクセス指定は、派生クラスを作る場合の基底クラスに対するアクセス指定でも成り立つ。→後述の・・）

一般にオブジェクトは、隠蔽する(カプセル化する)ことを前提に設計するので、**class** で**宣言する方が多い**。しかし、完全に隠蔽してしまう **private** では、再利用の範囲があまりにも狭まってしまう。
上記の PERSON クラスを継承して **STUDENT** クラス作り、名前と年齢を別の形式で表示しようとしても、基底クラスメンバが **private** では利用しようがない。そこで、隠蔽の度合いを緩めて派生クラスのメソッド内でアクセスを許すための **protected** 指定がある。これは、基底クラスの基本的な考えを継承するのであれば、利用を許すと言う考えで使われる。
例えば、右のように **protected** 指定した場合、下記のように **public PERSON** で継承した STUDENT のコンストラクタや disp メソッドで name や、age をアクセスできるようになる。

```
struct  PERSON {
protected:
    char name[64];//名前データメンバ
    int  age;     //年齢データメンバ
public:
    PERSON(char *s, int d);
    virtual void disp();//表示メソッド
};
```

```
//人を継承した学生クラス
class STUDENT : public PERSON{
    int  grade;  //成績(グレード)
public:
    //引数つきコンストラクタ定義
    STUDENT(char *, int , int );

    void disp();
}
```

所で、この **PERSON** の前に付く **public** は、基底クラスに対するアクセス指定と呼ばれ、メンバ宣言のアクセス指定とは異なり、継承する時に、基底クラス内の protected と public メンバのアクセス制限をそのまま制限で自身のメンバとする継承であることを指定しており、**public 継承**と言う。
対して、基底クラスの protected、public のメンバを、自身のメンバとして継承する時に private メンバに宣言し直すように継承する **private 継承**がある。以下にそのアクセス例を示す。

（このメンバ制限の継承を直接指定するアクセス宣言の規則もあるが、説明しない）

```
class PERSON {
protected:
    char name[64];
    int  age;
public:
    PERSON(char *s, int d);
    void disp();
};
class STUDENT : public PERSON{
protected:
    int  grade;
public:
    STUDENT(char *, int , int );
    void disp();
};
struct SpecialSTUDENT:public STUDENT
{//特別学生
    int scholarship; //奨学金
    SpecialSTUDENT();
};
```

public 継承なので name と age は protected のままで**利用可能**

```
class PERSON {
protected:
    char name[64];
    int  age;
public:
    PERSON(char *s, int d);
    void disp();
};
class STUDENT : private PERSON{
protected:
    int  grade;
public:
    STUDENT(char *, int , int );
    void disp();
};
struct SpecialSTUDENT:public STUDENT
{//特別学生
    int scholarship; //奨学金
    SpecialSTUDENT();
};
```

private 継承なので name と age は private になり**利用不可能。**しかし、STUDENT の disp メソッドで表示することはできる。→うまくメソッドを利用すれば参照可

（12）フレンドクラス、フレンド関数

プログラムを作成する場合、複数のクラスが絡み合って、作成される場合がよくある。その場合に相互にメンバを利用する頻度は大きいであろう。この複数のクラスが継承関係の場合は、基底クラスのメンバをプロテクトにすることで、派生クラスでは利用できるが、その逆をしたい場合にクラスをカプセル化したままだと、使えない。『メンバを利用するには public になければならない。』しかし、『ｵﾌﾞｼﾞｪｸﾄ指向として一般利用者に対するカプセル化（private）もしたい』こうしたジレンマが生じる。この回避策の一つとしてフレンドクラスの利用が挙げられる。その方法は、**friend** を伴ったクラス宣言をクラス内で宣言すればよい。

```
#include <stdio.h>
#include <string.h>
class  PERSON {//人クラス
    char name[64]; //名前データメンバ
    int  age;      //年齢データメンバ
    PERSON(char *s, int d);  //コンストラクタ
    virtual void disp();     //表示メソッド
    friend struct GROUP;     //フレンド宣言
};
PERSON::PERSON(char *s, int d) {//コンストラクタ定義
    strcpy(this->name,s);
    this->age = d;
}
void PERSON::disp(){//表示機能
    printf("名前:%8s /年齢%4d¥n",this->name, this->age);
}

struct GROUP {
    PERSON **array;   // array[0], array[1], と配列として使う。
    int capacity;     //上記配列の割り当て数
    int count;        //記憶データの個数
    GROUP(int n);
    void addPerson(char *s, int d);//追加
    void disp();
};
GROUP::GROUP(int n) { //コンストラクタ定義
    this->array = new PERSON *[n];
    this->count = 0;
    this->capacity = n;
}
void GROUP::addPerson(char *s, int d) {//追加メソッド
    array[count++] = new PERSON(s,d);
}
void GROUP::disp(){   //表示メソッド
    for(int i = 0; i < this->count; i++){
        this->array[i]->disp();
    }
}
void main()
{
    GROUP group = 100;        //100 個の要素を管理させる。
    group.addPerson("鈴木",24);       //データを追加設定
    group.addPerson("佐藤",20);
    group.addPerson("佐々木",19);
    group.disp();     //グループ全体の表示
```

例の **PERSON** は、public:がないので、全てのコンストラクタを含めたメンバが **private** になる。しかし、**friend** があるので、GROUP クラスのメソッド定義内では、全ての **PERSON** 内メンバをアクセスできる。

private メンバ関数なのにアクセスできる。

}

以上のように、**friend** を伴った宣言のクラス内では、クラスのアクセス制限を解除する働きがある。つまり、カプセル化の原則の例外を作る指定とも言える。それは、ｵﾌﾞｼﾞｪｸﾄ指向のカプセル化と言う保護壁を壊して特別の通路を作ってしまうと言う危険を伴うので、無闇に多用することは避けるべきであると言われる。なお、**friend** を伴った宣言はクラス以外にグローバル関数にも適用できる。次は、グローバル関数を **friend** した宣言した簡単な例で、A クラスと B クラスの両方でグローバル関数 subRoutine をフレンド宣言している。

```
class A{
   int a;
   friend void subRoutine();
};
```

```
class B{
   int b;
   friend void subRoutine();
};
```

クラス A、B ともコンストラクタや、データメンバが private であるが、右の friend 宣言された関数内では、生成やメンバ変更が自由に出来る。

```
void subRoutine()
{
      A a;
      B b;
      a.a = 1;
      b.b = 2;
      省略
```

もし単一クラスに対して、グローバル関数の **friend** 宣言を行いたい場合、それはクラス特有のグローバル関数になるので次の **static** メソッドにすることを薦める。

（13）static メンバ

static 宣言したメンバは、『ｵﾌﾞｼﾞｪｸﾄを指定しないでアクセスするメンバ』になる。言い方を変えると、static 宣言したメンバは、そのメンバを持つクラスの**全てのｵﾌﾞｼﾞｪｸﾄ**で共有されるメンバでになる。static 宣言は、データメンバにも適用できる。これらを public 指定したとすれば、それらはそのクラス指定を伴ってアクセスするグローバル関数とグローバル変数であると言える。
以下に、ｵﾌﾞｼﾞｪｸﾄの存在個数を自動的にカウントする COUNT クラスの例で使い方を示す。

```
#include <stdio.h>
#include <string.h>
class COUNT {
    static int count; //ｵﾌﾞｼﾞｪｸﾄ生成件数を数える スタティックメンバ
public:
    COUNT(){ count++; } //コンストラクタで計数
    static int getCount() { return count;}//生成件数取得スタティック関数メンバ
};
class  PERSON :private COUNT{//人クラス
    char name[64]; //名前データメンバ
    int  age;      //年齢データメンバ
public:
    PERSON(char *s, int d){ strcpy(name,s); age = d;} //コンストラクタ
};
int COUNT::count = 0;
void main()
{
    COUNT c1;      //１件目の生成でカウント値が１
    PERSON m1("鈴木",20); //２件目の生成でカウント値が２
    PERSON m2("山田",21);
    printf("生成件数:%d 件¥n",COUNT::getCount());

}
```

static メソッド内では、ｵﾌﾞｼﾞｪｸﾄのデータメンバ（this が持つメンバ）を使えない。ｵﾌﾞｼﾞｪｸﾄ指定なしで使うのだから、当然であるが・・

static メンバはグローバル変数と同じように外部で定義する。
そうしないと、『private: static int　COUNT::count″は未解決です』のリンクエラーになる。

static **メソッドの呼び出し方は、クラス名を指定する。**
例では COUNT **が１つとその継承ｵﾌﾞｼﾞｪｸﾄが２つあるので、**
『生成件数: ３件』が出力される。

ｵﾌﾞｼﾞｪｸﾄを指定して、**c1.getCount()** と実行してもよい。しかし、この例では private 継承なので PERSON::getCount()や c1.getCount()にするとエラーになる。

（14）抽象クラス

これまででオブジェクトをポインタで取り扱う場合で、しかも基底クラスのメンバ関数が仮想メンバ関数と呼ばれるvirtualの宣言を伴うものに限られて、メンバ関数が自動的に選択されて実行されることを示した。これは継承するクラスに、基底クラスの仮想関数と同じ名前、同じ戻り型、同じ種類の引数を持つメンバ関数を用意（オーバーライドと呼んだ）すると、そのメンバ関数も仮想メンバ関数になり、オブジェクトの種類に合ったメンバ関数が呼ばれる規則であった。

この考えを発展させて、派生クラスにおける仮想メンバ関数の定義を強制させるためだけのクラスを用意することがある。これは、派生クラスで作るべき機能を、基底クラスで予約してしまうもので**抽象クラス**と呼ばれる。クラスを抽象クラスにする方法は、メソッドを**純粋関数**と呼ばれるもので宣言すればよい。それには、次の例のⒶのようにvirtualの関数に『　**=0**　』を記述すればよい。

binary.h　内容

```
class Binary {
    char *array;//ﾊﾞｲﾅﾘｰｲﾒｰｼﾞの先頭ﾎﾟｲﾝﾀ
protected:
    void setThisPointer(void *p){array = (char *)p; } //読み書き位置ﾎﾟｲﾝﾀ設定
public:
    Binary() {} //ﾃﾞﾌｫﾙﾄｺﾝｽﾄﾗｸﾀ
    virtual int getSize() = 0;
    virtual void display() = 0;
    void write(FILE *fp){     //arrayの内容をファイルへ書き込む
        int size = setSize();//派生クラスより取得する読み書きサイズ
        fwrite(&size, sizeof(size), 1, fp);
        fwrite(array, size, 1, fp);
    }
    void read(FILE *fp){      //arrayの内容をファイルから読み込む
        int size;
        fread(&size, sizeof(size), 1, fp);
        fread(array, size, 1, fp);
    }
};
```

Ⓐ『　**= 0**　』があるので getSize と、display は**純粋関数**になり、純粋関数を含む**Binary**は**抽象クラス**になる。

抽象クラスでは、派生クラスに作る機能（純粋関数）を自身では定義せずに予約するだけなので実際のメソッド定義は派生クラスの方で行われる。そのため抽象クラスのオブジェクトを生成することはできない。（抽象クラスのオブジェクトを用意する宣言を行うとエラーになる。）

Binary bin;　　とした場合のVCでのエラーは、

『抽象ｸﾗｽあるいは構造体のｵﾌﾞｼﾞｪｸﾄが宣言されています』になる。

また、右記のように、Binaryの派生クラスとして、Stringを作成しても、GetSizeとDisplayﾒｿｯﾄﾞを定義しなければ、やはりエラーになる。

純粋関数を持っている抽象ｸﾗｽのｵﾌﾞｼﾞｪｸﾄを作ったエラー

```
#include "binary.h"

class String : public Binary{
public:
        char str[100];
        String(){}
};
void main()
{
        String s;
        ・・・省略・・・
```

つまり、純粋関数をオーバーライドして定義しない限り、何回継承しようと、純粋関数をもっているクラスは抽象クラスと言うことになる。言い方を変えると、binaryクラスは、クラスを継承してそのメンバ関数にgetSizeやdisplayﾒｿｯﾄﾞを**作らなければ、利用させないクラスである**！と言える。

抽象クラスでは、作られてない宣言だけの純粋関数名を利用したプログラミングができる。

前例の **Binary** クラスの目的は、バイナリーファイル操作を目的に作ったクラスで、自身のメンバにファイル保存機能の Write メソッドを用意しているが、派生クラスに予約した（自身では定義していない）getSize メソッドを利用して作っている。（派生クラスに getSize があることを前提に使っている。）

以下に、この Binary クラスの利用例を示す。

```
01  #include <stdio.h>
02  #include <string.h>
03  #include "binary.h"
04
05  class String : public Binary{
06  public:
07      char str[100];     //文字列
08      String(char *s=""){//コンストラクタ
09          strcpy(str,s);
10          setThisPointer(str);
11      }
12      int getSize() { return sizeof(str); }
13      void display() { printf("%s¥n", str); }
14  };
15
16  void main()
17  {
18      FILE *fp = fopen("test.dat", "rb");
19      String *s;
20      if(fp == NULL) {  //ファイルが存在しない時は、" abc "で新規作成
21          s = new String("abc");
22      } else {
23          s = new String;
24          s->read(fp);  //ファイルが存在する時は、ファイル内容を設定
25          fclose(fp);
26      }
27      s->display();      //確認の表示
28      printf("文字列="); scanf("%s",s->str);
29      fp = fopen("test.dat", "wb");
30      s->write(fp);
31      fclose(fp);
32  }
```

Binary クラスのファイル操作を利用するので継承する

①このクラスの保存位置（アドレスを設定）

②純粋メソッドをオーバーライドしてメソッドの実態を定義する。（virtual が無くても仮想関数である）

Binary クラスの write メソッドで保存している。この内部では String の getSize を使っている。

上記では、ファィル保存を Binary で行っているが、保存位置と保存サイズの情報は、派生クラスの String から取得している。保存位置は String のコンストラクタ内①の getThisPointer 実行で取得し、保存サイズは純粋関数をオーバーライドした String の getSize メソッド②より取得している。この『派生クラスのコンストラクタで保存位置を設定し、保存サイズを知らせる getSize をオーバーロードするだけで、read や write が使えるようになる』のパターンは、当然に例で示した String 型に限らず、任意の派生クラスで適用できる。

また、上記例では、String クラスの str をファイル読み書き対象にしただけであるが、これを発展させて、他の任意メンバ含めた（ポインタメンバでは意味をなさないが‥）オブジェクト全体を保存することも可能である。そのためには、読み込み時にオブジェクトを識別する仮想ﾃｰﾌﾞﾙへの**ポインタ**を壊さないように復元しなければならない。この**ポインタ**は **this** のポインタ先頭にあるので、**this** より**ポインタ**の記憶サイズだけ進ませたアドレスを setThisPointer で指定し、getSize では仮想ﾃｰﾌﾞﾙのポインタ分少ないサイズを知らせればよい。つまり次の変更で、任意のクラス全体をファイルへ保存し、ファイルから復元できることになる。

①の個所を変更→`setThisPointer((char *)this + sizeof(void *));`

②の個所を変更→`int getSize() { return sizeof(String)-sizeof(void *); }`

（15）演算子のオーバーロード（多重定義）

C++では、同じ関数名で異なる機能の関数を定義できた。その場合、引数の数や引数の型の違いに基づいて適切な関数が呼び出される。

右のように定義してあると、

print('0');　と実行すれば

putchar が実行され、

print("abcd");と実行すれば

printf が実行されることになる。

```
void print(int d)
{
        putchar(d);
}
void print(char *d)
{
        printf("%s",d);
}
```

そして、同じようなことが演算子にまで発展している。つまり演算子の機能まで多重に定義できる。それには、operator 予約語を付けて次のように演算子を定義するもので、演算子関数と呼ばれる。

```
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
class String{
public:
    char * pBuf;
    String() { pBuf = new char [256]; }
    ~String() { delete []pBuf; }
    const String &operator = (int val) { //整数の代入を可能に
        itoa(val,pBuf,10);
        return *this;
    }
    const String &operator = (const char *s);      //文字列の代入を可能に
    const String &operator = (const String &s) {
        *this = s.pBuf;              //上記文字列の代入演算を利用
        return *this;;
    }
    int operator ==(const char *s) const { return ! strcmp(pBuf,s); }
};
const String & String::operator = (const char *s) //①
{
    strcpy(pBuf,s);   //文字列のコピー
    return *this;
}
void main()
{
    String s1,s2,s3;
    s1 = 123;  //構造体(クラス)変数へ整数代入
    s2 = s1;   //①'内部で、文字列のコピーが行われる。
    printf("%s¥n",s2.pBuf);
    s1 = "XYZ";       //(char *)を代入すると、全体がコピーされる。
    printf("%s¥n",s1.pBuf);
    if(s1 == "XYZ") {              //②
        puts("一致");
    }
}
```

ここでは、**String** 型への=演算子による代入処理を、色々な型で可能なように、演算子多重定義、している。なお、上記の 123 代入演算は関数のように、**s1.operator =(123);**と書ける。 また、

" ZYX "の代入は **s1.operator =("XYZ");**と書ける。つまり、以下の左右は同じ意味になる。

ｵﾌﾞｼﾞｪｸﾄ. operator 演算子(引数);　⇔ ｵﾌﾞｼﾞｪｸﾄ 演算子 引数

例　　**s1.operator = ("xyz");　⇔ s1　　=　"xyz";**

この左の表現は、演算子の**関数呼び出し**と呼ばれ、表記的曖昧さをなくす場合などに使われる。

① の＝演算子関数の中で、文字列コピーが行われている。もし、演算子関数が定義されていないと、代入処理では、単なる構造体の代入で、pBuf のポインタ代入により、代入前に記憶していた new で作成した非億領域の管理情報を失うメモリーリークが起きてしまう。(→①'の個所で使っている)

(演算子関数が定義されていないと　s2 = s1;　は、s2.pBuf = s1.pBuf;　の意味になる。)

また、＝の演算子関数の戻り値はが、(String &)と参照型に統一されているが、これにより、
s3 = s2 = s1;　　の表記が可能になる。(s2 = s1 の演算結果で、s3 を設定)
なおここで、上記例において String s4 = "abc"; の表記ができないことを付け加える。
この表記の＝は演算子でなく初期化指定子である。よって、String に(char*)の変換コンストラクタがあってできる表記である。

さて、②の if 文の、**s1 == "XYZ"**　が可能なのは、比較演算子がオーバーロードしてあるためであるが、ここで **if("XYZ"==s1)** と順番を逆にしたらどうなるであろうか？
結論は、『二項演算子 '==' : 型 'class String' の右ｵﾍﾟﾗﾝﾄﾞを扱う演算子は定義されていません。』と言うエラーで不可能である。これは、**if("XYZ". operator ==(s1))** と書き換えるとすぐに理由は分かる。つまり、**"XYZ"** は String 型のｵﾌﾞｼﾞｪｸﾄではく、**operator** を含めたメソッドが実行できない訳である。これは、次のようなグローバルな演算子関数 **operator==()** を定義すればよい。

```
int operator ==(const char *s,const String &str)
{
        return ! strcmp(s,str.pBuf);
}
```

一般的にはクラスのメンバは非公開にする場合が多いので、freiend で宣言することが多い。

```
class String{
        ・・・省略・・・
        friend int operator ==(const char *s,const String &str);
};
```

これらの用法は単項演算子にも適用できる。また単項演算子、二項演算子のどちらにも使える演算子の場合は、それぞれの用途別にオーバーロードできる。++と--演算子に対しては前置、後置の２通りを定義できる。以下に String クラスに(cahr *)のキャスト演算子と、内部管理文字列を1 文字文だけ後方シフトしてずらす++演算子をオーバーロードした例を示す。

```
class String{
    ・・・省略・・・
    operator char *() {//キャスト演算子
        return pBuf;
    }
    char *operator ++() { //前置
        for(int n = strlen(pBuf); n >= 0; n--)
            pBuf[n+1]=pBuf[n];//一文字後方要素へシフトする繰り返し
        return pBuf;
    }
    char *operator ++(int) {//後置は、int 引数をダミーとして用意する形態で定義する
        for(int n = strlen(pBuf); n >= 0; n--)
            pBuf[n+1]=pBuf[n];//一文字後方要素へシフトする繰り返し
        return pBuf+1;
    }
};
```

```
s2 = s1 = "123";
*++s2 = 'A';
printf("%s¥n",(char *)s2);
*s3++ = 'A';
printf("%s¥n",(char *)s3);
```

結果

```
A123
1A23
```

このように演算子多重定義は、演算子の表記方法を人間の習慣に合った表記に拡張する便利な機構である。しかし当然に、本来の慣例的表現以外の意味を定義してしまうとプログラムの可読性を落とし、誤用が生じやすくなる。よって定義においては、対象用途を広域的視野から判断し、十分に検討した上で定義すべきである。**基本形だけを受け取る演算子のオーバーロードはできない**
なお、全ての演算子が多重定義できる訳でではなく、次の演算子などは多重定義できない。

.　　::　　?:　　sizeof

また、デフォルト引数の使用、オペランドの数や優先順位の変更は出来ない。

（16）例外処理

C++では、重大なエラーに対処するために、スコープ制限を越えたエラーハンドリング機構が備わっており、このメカニズムを例外（exception）と呼ぶ。手続きとしては、『throw を使った例外投入』、『try を使った例外確認範囲の設定』『catch を使った例外補足』の３種類を記述することになる。

まず、throw を使った例外投入は、一般にクラスや関数の設計者が使う。これは、自身が設計したクラスや関数の実行過程で、予期されるエラーをチェックし、エラーと判断される時 throw で値やｵﾌﾞｼﾞｪｸﾄを指定して例外として投入する。この時、何の**型**を投げるか（指定する）が重要になる。なぜなら、catch を使った例外補足では、この型の引数で受け取るからで、その型の catch を書かなかった場合、補足できなかったと言うことでプログラムがエラー終了することになる。

例として、指定ファイル名を引数にするコンストラクタを持つ FileObject クラスを次のように定義し、その読み取りオープンしようとするファイルがない場合の例を示す。

fileobject.h

```
struct FileObject{
    FILE *fp;
    FileObject(char *filename){
        fp = fopen(filename, "r");
        if(fp == NULL){
                throw "FileObject Exception";   ①
        }
    }
    ····省略····
};
```

（char *）のメッセージを投入している。

```
#include <stdio.h>
#include "fileobject.h"

void main()
{
    FileObject *pF;

    pF = new FileObject("data.txt"); //既存ファイルを内部でオープン

    //····省略····
}
```

上記の場合、FileObject のコンストラクタ内で既存のファイルをオープンしているが、存在しない時エラーになり、throw が実行されるので、『バンドルされてない例外』の実行エラー（異常終了）が起きる。

次に、この例外を補足し、異常終了を回避して、メッセージを出す main 関数を次に示す。

```
void main()
{
    FileObject *pF;

    try {
        pF = new FileObject("data.txt"); //既存ファイルを内部でオープン
    }
    catch(char *msg){ //②
      printf("%s¥n",msg);
    }
    catch(int d){   //③
      printf("Exception code:%d¥n",d);
    }
    //····省略····
```

try 内のエラー処理で throw されると、ここが実行され、throw された文字列の『FileObject Exception』が表示される。その後、残りのプログラムを継続する。

上記では、**catch（型 仮引数）｛　｝**が２つあるが、**型**が違えばいくつでも記述できる。

この例では、fileobject.h の中の①で、"FileObject Exception"と言う(char *)型を投入している。よって、②の **catch** 内が実行される。もし、①で **throw　200；**と整数を投入した場合は、③の **catch** 内が実行される。そして、投入したデータの型を **catch** する部分を作らなかった場合は、異常終了することになる。

（17）C++のストリーム

下記でC言語の標準関数と、C++の入出力を比較する。左右はほぼ同じ実行結果になる。

```
#include <stdio.h>
void main()
{
   int i;
   double d;
   char s[256];

   fprintf(stdout,
        "整数,実数,文字列入力=");
   fscanf(stdin,"%d %lf %s",
        &i, &d, s);
   fprintf(stdout,
        "%d,%f,%s が入力された¥n",
        i,d,s);
}
```

```
#include <iostream.h>
void main()
{
   int i;
   double d;
   char s[256];

   cout << "整数,実数,文字列入力=";

   cin >> i >> d >> s;

   cout << i << ',' << d << ',' << s
        << "が入力された¥n";

}
```

C言語では、画面などに出力する場合に **stdout** の標準出力先という(FILE*)型を指定したが、C++では、**cout** と言う(ostream)型を指定する方法がある。**cout** は、標準出力ストリームと呼ばれ、<<の演算子を使って出力したいデータを指定する。
<<は、ostreamクラスで、様々な型に対し、戻り値が（ostream&）になる演算子多重定義されていおり、**cout** << i << ',' << d << ',' << s　と、並べた順番に出力できる。

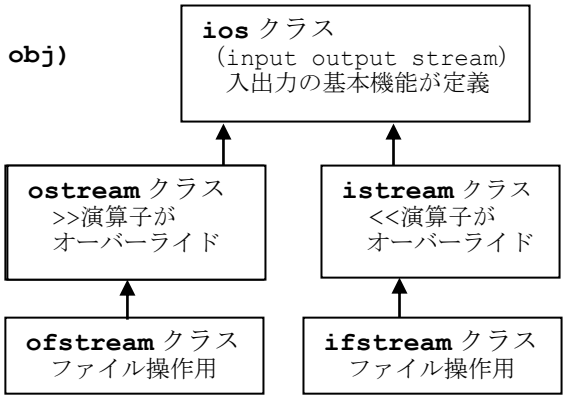
```
          ┌──┐
          i を出力し、結果は cout と同じ（ostream&)型
     ┌──────────┐
     結果は cout と同じ（ostream&)型
┌──────────────────┐
```

同様に、C言語の標準入力である **stdin** に対応して、C++では **cin** が入力ストリームに使え、<<演算子で変数を指定する。**stdin** や、**cin** は、**iostream.h** のインクルードで使えるようになる。

これらは、自作クラスを定義する場合、次のように使われる。

```
#include <iostream.h>
class IDS {// intとdoubleとchar配列をまとめた確認用クラス
   int i;
   double d;
   char s[256];
   friend ostream& operator<<(ostream& ,const IDS & );
   friend istream& operator>>(istream& , IDS & );
};
ostream& operator<<(ostream& os,const IDS & obj)
{
     os << obj.i << ',' << obj.d << ',' << obj.s << "が入力された¥n";
     return os;
}
istream& operator>>(istream& is, IDS & obj)
{
     is >> obj.i >> obj.d >> obj.s;
     return is;
}
void main()
{
   IDS ids;
   cout << "整数,実数,文字列入力=";
   cin >> ids;
   cout << ids;
}
```

右記のような構造でファイルを扱うクラスもある⇒



（18）テンプレート

C++のテンプレート機能を利用すると、まだ型が決まっていないパラメータを持つ関数やクラスを定義することができる。テンプレートとは、型板のことで**テンプレート**と言う未定義の型を使って定義しておいて、使用する時の宣言や呼び出しで型が特定されるメカニズムになっている。次で、関数テンプレートの具体例で説明する。

```
#include <stdio.h>

template <typename T> T abs(T a)
{
        return a < 0 ? -a: a;
}

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

void main()
{
        printf("%d¥n", abs(1));
        printf("%f¥n", abs(5.123));

        printf("%d¥n", big(1,5));
        printf("%f¥n", big(5.123,6.123));
        printf("%f¥n", big(5.123F,6.123F));
}
```

typename の代わりに class キーワードが使われた時期があり、
それでも同じように定義できる。

まず上記の **template**<**typename T**>所で、**T** と言う名前のテンプレート型を一時的に定義している。そしてこの一時的な T と言う型があるとして、その型を引数で使ったり、戻り値の型に使ったりして、abs や big の関数を定義している。すると、関数呼び出しで abs(1)とした場合、引数が 1 の int 型なので、T の型の代わりに引数と戻り値が int 型の定義であるとして、それが呼び出されることになる。同様に abs(5.123)と呼び出した場合は、引数が 5.123 と double 型なので、引数も戻り値も double 型の abs が呼び出されることになる。このように、型を未定儀とし、呼び出し側で型を特定する関数を**汎用関数**と呼ぶ。

なお、引数の数が複数ある big 関数の呼び出しには注意が必要になる。それは、複数で同じテンプレート型で定義しているため、例えば printf(″%d¥n″, big(5,5.1));のように異なる型を引数にすると、int 型と double 型のどちらをテンプレート型の代わりに使うかあいまいで特定できなくエラーになってしまう。

また、汎用関数は明示的に特定の型でオーバーロードすることができ、その型についてはカスタマイズした関数として定義が可能になる。例えば、上記例の最後の float を引数にした場合だけ、追加メッセージとして『明示的関数の実行』の出力を埋め込みたい場合は次のように追加定義すればよい。

```
template<> float big<float>(float a,float b)
{
        printf("明示的関数の実行:");
        return a < b ? b : a;
}
```

次にクラスへテンプレートを適用してみる。すると関数と同様に、変数を宣言する時点でメンバの型を特定させるクラスを定義することができ、**汎用クラス**と呼ばれる。次の例では配列を管理する簡単な Array と言うテンプレートクラスを定義している。

```
template <typename T>class Array
{
	T *array;
	int mMaxIdx;
public:
	Array() { mMaxIdx = 100;array = new T[mMaxIdx]; }
	virtual ~Array() { delete []array;  }
	void Set(int idx,T d) {  array[idx] = d; }
	T GetAt(int idx){ return array[idx]; }
	void Insert(int n, T d);
};

template <typename T> void Array<T>::Insert(int n, T d)
{
	for(int i=mMaxIdx-2; i >= n; i--)
		array[i+1] = array[i];
	array[n] = d;
}


void main()
{
	Array<int> AI;    ←①
	Array<char> AC;   ←②

	for(int i = 0; i < 10; i++){
		AI.Set(i,'0'+i);
		AC.Set(i,'0'+i);
	}

	AC.Insert(3,'A');
	AC.Insert(3,(int)'A');

	for(i = 0; i < 10;i++){
		printf("%d='%c'¥n",AI.GetAt(i),AC.GetAt(i));
	}
}
```

実行結果

```
48='0'
49='1'
50='2'
65='A'
51='3'
52='4'
53='5'
54='6'
55='7'
56='8'
```

main 関数の①と②で、int 型要素の配列を管理する AI と言うオブジェクトと char 型要素の配列を管理する AC と言うオブジェクトを用意している。このように、汎用クラスのオブジェクト生成は、< >の中で*特定の型*を記述しなければならない。そしてオブジェクトの生成時点で、テンプレートで宣言した一時的な型 T の変わりにこの*特定の型*を使ったクラスと言うメカニズムでオブジェクト生成される。よって< >の中での指定する無数のバリエーションが生まれる。しかもこのクラスで定義されるメンバ関数が、それら無数のクラスで同じように使える訳で、より複合的で抽象的役割を意味つけたクラスを作ることができる。

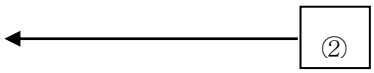
（19）STL(Standard Template Library)

C++では、テンプレートを利用した STL と呼ばれるライブラリが用意されており、それにはよく使用されるデータ構造とアルゴリズムが用意されている。STL の基本要素としては、**コンテナ、アルゴリズム、反復子**の三に分けられる。一般にプログラムする場合は、管理するデータへの操作やその性質よりデータ構造を決定する訳であるが、そのデータ構造の種類として、配列なら vector コンテナ、リスト構造なら list コンテナ、キューの構造なら deque コンテナ、連想配列なら map コンテナを利用することになる。ここで言うコンテナとは、そのデータ構造の要素を管理するための汎用クラスで作られたオブジェクトを意味する。

また各コンテナに対してアルゴリズム(検索やソートやマージ）が汎用関数の形態で用意されている。それには、反復子（iterator;イテレータ）と呼ばれる操作用のクラスを利用して、各種コンテナに対し一連の操作方法が提供されている。以下この汎用クラスを模擬するクラスを作って、理解を深めてみよう。これは実際の STL とは全然違う中身であるが、表面的に少し似せた List コンテナ用クラスと反復子用の iterator クラスを実装し、main で確認した例である。

```
#include <stdio.h>
template <typename T>class List{
public:
    List() { }
    virtual ~List() { Top.pNode = NULL; }

    class _Node{ //リスト構造のノード定義
    public:
        T data; //管理するデータ
        _Node *pNext;
    public:
        _Node() { pNext  = NULL; }
    };

    class iterator{//反復子定義
    public:
        _Node *pNode;
        iterator(){ pNode = NULL;}
        bool operator=(iterator &it){ return pNode = it.pNode; }    //代入
        bool operator==(iterator &it){ return pNode == it.pNode; }//比較
        bool operator!=(iterator &it){ return pNode != it.pNode; }//(否定)
        T operator*(){ return pNode->data; }       //間接
        iterator &operator++(int){//後置++
                pNode = pNode->pNext;
                return *this;
        }
    }Top,End;                       //リスト先頭と最後
    void push_front(T t);
    iterator begin() { return Top; }
    iterator end() {  return End; }
};
template <typename T> void List<T>::push_front(T it)
{
        _Node *pNewData = new _Node();
        pNewData->data = it;
        pNewData->pNext = Top.pNode;
        Top.pNode = pNewData;
}
```

```
void main()
{
        List <int> lst;
        lst.push_front(4);
        lst.push_front(3);
        lst.push_front(2);
        lst.push_front(1);

        List <int>::iterator p = lst.begin();
        while(p!= lst.end()){
                printf("%d¥n",*p);
                p++;
        }
}
```



①で int 型データを記憶するリスト構造のコンテナオブジェクト lst を用意し、そのリスト構造の最初に push_front メソッドでデータ 4,3,2,1 を追加している。

②で反復子 p を用意しているが、初期設定として lst.begin()で得られるリスト先頭要素を指す初期化を行っている。これは、=演算子をオーバーライドすることで実現されている。同様に!=や==も反復子が管理するデータが同じものかを判定できるようにオーバーライドされている。

また反復子を p++すると、反復子 p が管理するデータが次のデータに移動するように++演算子がローバーライドされている。同様に*演算子も反復子が管理されてデータを取り出すように、オーバーライドされている。繰り返し条件としてコンテナの end()メソッドで得られるデータを比較しているが、これは反復子を使う場合の定石で end()メソッドが最後のデータを指し示す反復子を返すものでなく、最後を越えて進み、なにも指し示してない反復子に達している状況と言う判定を行わせていることに注意されたい。さて、汎用クラスとして作成しているので<>の中で指定する任意の型がそのままリストで管理できることになる。しかもこのような表現は、他のコンテナクラスでも同じような取り扱いができる。以下に実際の STL を利用した list（リスト）と vector（配列）のコンテナを利用した例を示す。なお、配列では先頭に挿入する push_front メソッドが用意されていないので、最後に追加する push_back メソッドを使っている。また、list では sort メソッドを呼び出して並び替えているが vector には存在しない（アルゴリズムの sort 汎用関数は使える。）なお、どちらの場合も std の namespace 指定が必要になる。

```
#include <stdio.h>
#include <list>
using namespace std;

void main()
{
  list <int> lst;
  lst.push_back(3);
  lst.push_back(7);
  lst.push_back(2);
  lst.push_back(5);
  lst.sort();

  list    <int>::iterator    p    =
lst.begin();
  while(p!= lst.end()){
       printf("%d¥n",*p);
       p++;
  }
```

```
#include <stdio.h>
#include <vector>
using namespace std;

void main()
{
  vector <int> vct;
  vct.push_back(3);
  vct.push_back(7);
  vct.push_back(2);
  vct.push_back(5);
  //   lst.sort();

  vector<int>::iterator p = vct.begin();
  while(p!= vct.end()){
       printf("%d¥n",*p);
       p++;
  }
}
```

（20）STL(Standard Template Library)の全貌

STL で定義されるコンテナを示す。

コンテナ名	説明	必要ヘッダー
deque	両端から操作可能	<deque>
list	線形リスト（双方向リスト）	<list>
map	キーと値をペアで格納する。 一つのキーが一つの値に対応	<map>
multimap	キーと値をペアで格納する。 一つのキーが複数の値に対応する場合あり	<map>
multiset	要素の重複が許される集合	<set>
priority_queue	優先順位付きキー	<queue>
queue	キュー（待ち行列）	<queue>
set	要素の重複が許されない集合	<set>
stack	スタック	<stack>
vector	動的な配列	<vector>

これらコンテナによって要素を格納する方式が異なるため、プログラムによって希望処理に合ったコンテナを選択しないとパフォーマンスが得られない。例えば vector の中央に要素を挿入する時間は線形的増大するが、list への挿入は一定の時間で完了する。つまり選択実装したコンテナによりプログラムの実行速度が大きく変わる可能性がある。また vector で使われる反復子は**ランダムアクセス反復子**と呼ばれ、++, --, -, +, []演算子を適用できるが、list で使われる演算子には、-, +, []がサポートされずに**双方向反復子**と呼ばれる。その他にも++演算子だけサポートされる前方反復子や入力反復子や出力反復子がある。map、multimap で使われる反復子は list と同じ双方向反復子になっているが、vector や list のシーケンスコンテナと異なり連想コンテナと呼ばれ、

pair のデータを使う反復子になっている。以下に簡単例を次に示す。

文字のキーで整数の入力後に、文字から一緒に記憶した整数を表示するプログラムである。iterator が first, second のペア情報になる。

```
#include <stdio.h>
#include <map>
using namespace std;
void main()
{
        map<char, int> m;
        char c;
        int i;

        for(;;){
                printf("入力 (文字 整数)>");
                if(scanf(" %c %d",&c,&i) != 2) break;
                m.insert(pair<char , int>(c,i));
        }
        map<char , int>::iterator p;
        printf("入力 (文字)>");scanf("%*s %c",&c);
        p = m.find(c);
        if(p == m.end()){
                printf("見つからない!!¥n");

        } else {
                printf("%c の整数は%d です¥n",c, p->second );
        }
}
```

さて、各コンテナにサポートされるメンバ関数を使う以外に、**関数テンプレート**を使ったアルゴリズムが数多く用意されている。これを使う必要性として、『同時に２つの異なる型のコンテナを扱える』という点が挙げられる。ますアルゴリズムを分類してみよう。国際標準のカテゴリ分類では、

『シーケンス非変更操作』、『シーケンス変更操作』、『ソート関連操作』になっているが、大さっぱすぎるのでそれぞれにN、C、Sの記号つけて以下に細分化してみる。

コピー操作			
copy　指定範囲を、第３引数へコピー	C	fill, fill_n　指定範囲と値で埋める。	C
copy_backword　後方よりコピー	C	swap_ranges　特定範囲要素の交換	C
iter_swap　２つの反復子が示す内容交換	C	swap　２つの値を交換	C
ソートされていないシーケンスの検索(ソートしていても使えないことはない)			
adjacent_find　隣接して同じ要素を検索	N	equal　２つの範囲が等しいか	N
find 特定範囲より値で検索、反復子を返す	N	find_end 部分シーケンス検索、末尾を返す	N
find_if ユーザー関数で true を返す要素検索	N	find_first_of 特定範囲要素の検索	N
mismatch 最初に相違する位置を検索	N		
ソート済みシーケンスの検索			
binary_search　２分探索	S	lower_bound 指定以上の最初の要素検索	S
upper_bound　指定以下の最後の要素検索	S	equal_range 挿入可能な範囲検索	S
ソート			
nth_element　特定範囲の最小要素反復子取得	S	sort　範囲をソート	S
stable_sort　等しい要素順を変えないソート	S	partial_sort 部分ソート	S
partial_sort_copy 部分ソート＆コピー	S		
要素の置換と削除			
remove 特定範囲から特定要素を削除	C	replace　特定範囲から特定要素を置換	C
remove_if　ユーザー関数が true になる要素削除	C	replace_if ユーザー関数が true の要素置換	C
remove_copy 特定範囲をコピー削除	C	replace_copy　特定範囲をコピー置換	C
remove_copy_if　ユーザー関数使用の削除コピー	C	replace_copy_if ユーザー関数使用の置換削除	C
unique 重複要素の削除	C	unique_copy　コピー（重複要素の削除）	C
シーケンスの変換と生成			
generate 生成関数が返す値を特定範囲に設定	C	generate_n　生成関数が返す値で n 個へ設定	C
transform 引数付き生成関数で、特定範囲に設定	C		
シーケンス並び順の変更			
rotate　特定範囲を左方向へ回転	C	rotate_copy 左方向へコピーしながら回転	C
reverse 範囲並び順の逆転	C	reverse_copy　範囲並び順をコピーと共に逆転	C
partition ユーザー関数が true の要素を先頭へ	C	stable_partition 相対的な並び順は維持	C
random_shuffle ランダムな並び変え	C		
ソート済みシーケンスのマージ			
merge　２つを指定シーケンスへマージ	S	inplace_merge 単一シーケンス内マージ	S
集合演算			
includes　一方が他方に含まれるかの判定	S	set_difference ２つの集合の差集合生成	S
set_intersection　交差集合（共通要素）の生成	S	set_symmetric_difference 一方だけ含まれる	S
set_union　２つの集合の和集合生成	S		
順列			
next_permutation シーケンスの次の順列を構築	S	prev_permutation シーケンスの前の順列構築	S
ヒープ演算(ルートを最初の要素とする木構造)			
make_heap　シーケンスからヒープを構築	S	pop_heap　指定の最初と最後-1 を交換し再構築	S
push_heap　要素をヒープ末尾に追加	S	sort_heap　特定範囲のヒープをソートする	S
最小値と最大値			
max　２つの値で大きい方を返す	S	max_element　最大要素への反復子を返す	S
min　２つの値で小さい方を返す	S	min_element　最小要素への反復子を返す	S
その他			
count　シーケンス内要素数を返す	N	count_if　ユーザー関数が true の要素数を返す	N
for_each　ユーザー関数を特定範囲に適用	N	lexicographical_compare　ｱﾙﾌｧﾍﾞｯﾄ順比較	S