Ⅲ オブジェクト指向プログラミング

学習目標
① オブジェクト指向プログラミング言語の種類を理解させる。
② Ｃ＋＋言語でのオブジェクト指向の考え方を理解させる。
③ Ｃ＋＋言語でオブジェクト指向を行うときの方針を理解させる。

1 オブジェクト指向プログラミング言語の種類

１９８０年代以降、オブジェクト指向の考え方を取り入れ、それを具体化するためにさまざまなオブジェクト指向プログラミング言語が設計された。例えば、Ｓｍａｌｌｔａｌｋ、Ｏｂｊｅｃｔｉｖｅ Ｃ、Ｅｉｆｆｅｌ、Ｃ＋＋、Ｊａｖａなどがそれにあたる。

これらの言語のうち、現在最も有力なのは、Ｃ＋＋言語である。Ｃ＋＋言語は、Bjarne Stroustrupによって１９８６年に提唱された言語である。その名前に＋＋というＣ言語の演算子が付加されていることからもわかるようにＣ＋＋というのは、Ｃの内容に１を加えたもの、すなわち、Ｃをより良いものに進化させたものという意味である。すなわち、Ｃ＋＋言語は、既に存在したＣ言語の文法を拡張し、オブジェクト指向言語へと成長させたものである。

Ｃ言語は、もともとは手続き型のプログラミング言語であり、ＵＮＩＸといったオペレーティングシステム（ＯＳ）から一般的なアプリケーションまで幅広く記述することができる言語として、大変多くの利用者を有している。

一方、Ｃ＋＋言語は、Ｃ言語と基本的に上位互換性をもたせて設計された。そして、Ｃ言語からの言語仕様の拡張によって、プログラムの実行速度やメモリ効率といった性能が著しく低下することがないことを目標に設計された。そのため、Ｃ＋＋言語として追加された新たな機能をあえて使わず、改良されたＣ言語として使用する場合、性能はそれほど低下しないようになっている。また、Ｃ＋＋言語をコンパイルするためのコンパイラも、Ｃ言語とＣ＋＋言語のどちらでも使用できるようにしているものが大半である。すなわち、Ｃ＋＋言語用のコンパイラを使用しているにもかかわらず、Ｃ言語と同じ手続き型のプログラミングを行うことも可能になっている。

そのため、多数のＣ言語ユーザが容易にＣ＋＋言語に取り組む環境は整っているということができる。

そこで、本書では、Ｃ＋＋言語をオブジェクト指向プログラミング言語の代表として取り上げることとする。

その際に注意しなければならないことは、Ｃ＋＋言語でのオブジェクト指向の考え方をしっかり理解しておかなければならないということである。そうしないと、Ｃ＋＋言語は使ってみたが、ちっともオブジェクト指向プログラミングにはならなかったといった事態が発生してしまう。

例えば、リストⅢ－１とリストⅢ－２を比較してほしい。

```
#include <stdio.h>
void main(void)
    {
    /* C言語によるあいさつ */
    printf("こんにちは");
    }
```

リストⅢ－１　C言語によるあいさつ

```
#include <iostream.h>
void main(void)
    {
    // C++言語によるあいさつ
    cout << "こんにちは";
    }
```

リストⅢ－２　C++言語によるあいさつ

リストⅢ－２は、C++言語の文法を使って記述してはいるが、本質的にはC言語の文法を使って記述したリストⅢ－１と何ら変わらない。C++言語を、ただ改良されたC言語としてしか使っていない。常に、C++言語でオブジェクト指向プログラミングをしているのだということを忘れないことである。

実をいうと、オブジェクト指向プログラミングを学ぶときには、これまでの手続き型プログラミングの知識がないほうがかえってスムースにいくことが多い。それは、生まれて初めてオブジェクト指向の考え方を学ぶ人の方が、手続き型プログラミングの経験に基づく先入観がなく、オブジェクト指向の考え方に、より自然に取り組むことができるからである。しかし、手続き型プログラミングの知識が全く無駄になるわけではない。オブジェクト指向の考え方は、プログラミングの複雑さを取り除くという目標を実現するために使用するのであり、あくまでもユーザの利益のためにあるものである。手続き型プログラミングに慣れたユーザにとっては、発想の転換によって、これほどにもプログラミングが簡単にできるようになるのかという落差を楽しむ気持ちが要求される。

また、一般的にC++言語とC言語とを比較した場合、C言語よりもC++言語の方がより大規模なプログラムまで記述することができる。また、同じ目的のソフトウェアを作る場合、C言語よりもC++言語の方がより短期間にプログラムを記述することができる。これも、実際にオブジェクト指向の考え方による設計やプログラミングを体験してみると実感できるはずである。

2　C++言語でのオブジェクト指向の基本事項の実現

ここでは、C++言語において、オブジェクト指向の考え方がどのように実現されているかについて、基本的事項に限り説明する。

(1) クラス

オブジェクト指向の考え方では、世界に存在するものは、クラスを用いて分類され、表

現される。そのため、C＋＋言語には、classという予約語が用意されている。そして、classで定義するのは型である。ここで定義する型というのは、整数型や文字型といった、あらかじめシステムが用意している型と同様のものである。

例えば、テレビ（Television）というクラスを考える。そして、テレビには、電源スイッチ、チャンネル、音量ボリュームという３種類の操作が指定できるものとする。そのとき、テレビというクラスをリストⅢ－３のように定義してみる。

```
class Television                          // ①テレビクラス
    {
    protected:                            // ②外部には非公開の部分
        int   iPower;                     //   ③電源
        int   iChannel;                   //   ④チャンネル
        int   iVolume;                    //   ⑤音量ボリューム
    public:                               // ⑥公開の部分
        Television(void);                 //   ⑦コンストラクタ
        ~Television(void);                //   ⑧デストラクタ
        void  DisplayProgram(void);       //   ⑨番組を画面に表示する
        void  DownVolume(void);           //   ⑩音量を下げる
        int   GetChannel(void);           //   ⑪チャンネルを知る
        int   GetPower(void);             //   ⑫電源スイッチのオン・オフを知る
        int   GetVolume(void);            //   ⑬音量を知る
        void  SetChannel(int iNumber);//   ⑭チャンネルを設定する
        void  SwitchPower(void);          //   ⑮電源スイッチをオン・オフする
        void  UpVolume(void);             //   ⑯音量を上げる
    };
```

リストⅢ－３　クラス定義の例（テレビクラス）

クラスの細かい部分については、これから説明していくことにするが、とりあえず重要なのは、③～⑤の部分はデータ部、⑦～⑯の部分は関数部であるということである。

従来のプログラミング手法では、まずデータ構造を定義した後、アルゴリズムを付け加える形でソフトウェアを開発していた。そのため、データ構造とアルゴリズムは、ばらばらに扱っていた。

一方、オブジェクト指向プログラミング手法では、まずクラスというデータ型の中で、データ構造とそれに対する操作を定義した後、それを用いてアルゴリズムを記述するという形でソフトウェアを開発する。そのため、データ構造とそれに関係するアルゴリズムを一体でとらえることができる。

特に、C＋＋言語の場合、クラスは、C言語における構造体を拡張し、構造体に関数まで含めたものということができる。これは、これまでテレビに関するデータと関数を、プログラムの別々の部分にばらばらに記述していたのをやめ、代わりに、テレビに関するデータも関数も、一つのクラスの内部に一緒に記述することを意味する。これによって、テレビというものをひとまとまりのオブジェクト（もの）として捉えることができる。

C＋＋言語では、クラスで定義された関数を「メンバ関数」と呼んでいる。メンバ関数は、オブジェクトに対してメッセージが送られたときに起動される。リストⅢ－３のテレビクラスの場合、⑦～⑯の１０種類のメンバ関数が定義されているが、これらはすべて電源、チャンネル、音量ボリュームという三つのデータを操作し、テレビというオブジェク

トの振る舞いを実現する。ちなみに、オブジェクト指向の一般的な用語では、メンバ関数のことを「メソッド」と呼んでいる。

また、Ｃ＋＋言語では、クラスで定義されたデータを「データメンバ」と呼んでいる。一方、オブジェクト指向の一般的な用語では「インスタンス」と呼んでいる。

次に、メンバ関数の実際の処理について説明する。メンバ関数の定義と実際の処理とは、別々に記述するのが普通である。メンバ関数の実際の処理部の例をリストⅢ－４に示す。

```
void  Television::Television(void)
    {                                   // コンストラクタ
    iPower=-1;                          // 電源スイッチをオフに設定する
    iVolume=5;                          // 音量ボリュームを５レベルに設定する
    iChannel=1;                         // チャンネルを１に設定する
    }

void  Television:: Television(void)
    {                                   // デストラクタ
    if(iPower==0)                       // 電源スイッチがオンの場合
        iPower=-1;                      // 電源スイッチをオフに設定する
    }

void  Television::DisplayProgram(void)
    {                                   // 番組を画面に表示する
    if(0==GetPower())                   // 電源スイッチがオンの場合
        {
        cout << "これは" << GetChannel() << "チャンネルです。¥n";
        cout << "音量は" << GetVolume() << "です。¥n";
        }
    }

void  Television::DownVolume(void)
    {                                   // 音量を下げる
    if(0==GetPower())                   // 電源スイッチがオンの場合
        iVolume--;
    }

int   Television::GetChannel(void)
    {
    if(0==GetPower())                   // 電源スイッチがオンの場合
        return iChannel;                // チャンネル番号を返す
    else                                // 電源スイッチがオフの場合
        return 0;                       // ０を返す
```

```
    }

int   Television::GetPower(void)
    {
    return iPower;                    // 電源スイッチを返す
    }

int   Television::GetVolume(void)
    {
    if(0==GetPower())                 // 電源スイッチがオンの場合
        return iVolume;               // 音量ボリュームを返す
    else                              // 電源スイッチがオフの場合
        return 0;                     // ０を返す
    }

void  Television::SetChannel(int iNumber)
    {
    if(0==GetPower())                 // 電源スイッチがオンの場合
        iChannel=iNumber;             // チャンネル番号を設定する
    }

void  Television::SwitchPower(void)
    {
    if(iPower==0)                     // 電源スイッチがオンの場合
        iPower=-1;                    // 電源スイッチをオフにする
    else                              // 電源スイッチがオフの場合
        {
        iPower=0;                     // 電源スイッチをオンにする
        cout << "テレビの電源がオンになりました。¥n";
        }
    }

void  Television::UpVolume(void)
    {                                 // 音量を上げる
    if(0==GetPower())                 // 電源スイッチがオンの場合
        iVolume++;
    }
```

リストⅢ－４　メンバ関数の実際の処理部の例（テレビクラス）

ここで、メンバ関数の名前の前にTelevison::という文字列が付加されているが、これは

テレビクラスのメンバ関数の処理を記述した部分であることを示している。また、特徴的なのは、③～⑤の三つの内部データを操作するために、Getxxxx及びSetxxxxというメンバ関数を用意していることである。これらの関数によって、外部から外部に対して非公開の内部データへ直接アクセスされることを防止することができる（実際のプログラミングでは、このメンバ関数の中で入力値の正当性のチェックを行う。すなわち、負の値が入力されたときは、エラーメッセージを表示する。しかし、本書では、煩雑さを避けるために、入力値の正当性のチェックに関する箇所は省略している。）。

また、メンバ関数の実際の処理部の中では、同じクラス内の他のメンバ関数を利用することができる。そして、その場合はメンバ関数の名前の前には何も付ける必要はない。しかし、どうしてもそのメンバ関数が、同じクラス内の関数であることを示したいときは、thisという予約語を使用して明示的に指定することができる。その指定方法は、次の通りである。

this->メンバ関数名(引数の並び);

⑦のコンストラスタ及び⑧のデストラクタについては、後でまとめて説明する。

(2) オブジェクト

Ｃ＋＋言語では、ユーザが定義したクラスから無数のオブジェクトを生成することができる。そして、生成の方法には２種類ある。

第１の方法は、オブジェクトそのものを作成する方法である。リストⅢ－５に示すように、直接オブジェクトを生成できる。この場合、オブジェクトはポインタ型ではない。

```
void main(void)
    {
    Television TV;          // TVというオブジェクトを生成しておく
    TV.SwitchPower();       // 電源スイッチをオンにする
    TV.DisplayProgram();    // １チャンネルの番組を画面に表示する
    }
```

リストⅢ－５　オブジェクトの生成例（１）

```
void main(void)
    {
    Television *TV;         // TVというオブジェクトへのポインタを用意する
    TV=new Television;      // TVというオブジェクトを新たに確保する
    TV->SwitchPower();      // 電源スイッチをオンにする
    TV->DisplayProgram();   // 番組を画面に表示する
    delete TV;              // TVというオブジェクトを解放する
    }
```

リストⅢ－６　オブジェクトの生成例（２）

第２の方法は、必要なときに動的にオブジェクトを作成する方法である。そして、不要になったらそのオブジェクトを削除する。その場合、クラスへのポインタ型を用意し、newという演算子を使ってオブジェクトを作成、deleteという演算子を使ってオブジェクトを削除する。リストⅢ－６に第２番目のオブジェクトの生成方法を示す。

new演算子でオブジェクトを作成した場合、そのメンバ関数を呼び出す方法が２種類ある。１番目が、

オブジェクト名->メンバ関数名(引数の並び);

２番目が、

(*オブジェクト名).メンバ関数名(引数の並び);

という方法である。どちらも全く同じ意味であるが、１番目の方法のほうが、オブジェクトのメンバ関数にメッセージを送っているという雰囲気が伝わってくるので、より望ましいのではないかと考える。

ところで、リストⅢ－５もリストⅢ－６も、実行結果は同じである。実行結果を図Ⅲ－１に示す。

```
テレビの電源がオンになりました。
これは1チャンネルです。
音量は5です。
```

図Ⅲ－１　リストⅢ－５とリストⅢ－６の実行結果

(3) 情報の抽象化

既に説明したように「情報の抽象化」は、「情報のカプセル化」や「情報隠蔽」といった用語で置き換えられることもある。Ｃ＋＋言語においては、クラス内部の各データ、メンバ関数が３種類のアクセス属性をもっている。そして、アクセス属性を適切に設定することによって、クラスの外部から勝手にクラス内部の情報を変更したり、参照したりできないようにすることができる。

Ｃ＋＋言語のアクセス属性は、private（私的属性）、protected（保護属性）、public（公的属性）の３種類である。まず、privateは、クラス内部からだけしかアクセスできないことを意味する。三つのアクセス属性のうちで、最も私的な性格の強い属性である。publicは、クラスの内部からでも外部からでも自由にアクセスができることを意味する。三つのアクセス属性のうちで、最も公的な性格の強い属性である。そして、protectedは、privateとpublicの中間に位置するクラスである。これは、クラス内部から、あるいは、クラスの外部でも自らの子クラスからだけはアクセスができることを意味する（子クラスについては、継承のところで説明する。）。

また、Ｃ＋＋言語では、構造体の内部は、何も指定しないとpublicのアクセス属性をもつようになっている。これにより、Ｃ言語との互換性が維持されている。一方、クラスの内部は、何も指定しないとprivateのアクセス属性をもつ。これにより、情報がクラスという入れ物、すなわちカプセルの中に隠蔽される。アクセス属性を表Ⅲ－1に示す。

ただし、ここで注意しなければならないのは、Ｃ＋＋言語のアクセス属性を適切に設定するのが、プログラミングをする人の責任であるということである。何も考えずプログラ

ミングをすると、オブジェクト指向の考え方に反して、クラスの外部から勝手にクラス内部の情報を変更したり参照したりすることができてしまう。

表Ⅲ－1　アクセス属性

アクセス属性	クラス外部からアクセス	子クラスからアクセス	クラス内部からアクセス
private	×	×	○
protected	×	○	○
public	○	○	○

(4) 継承

継承の機能は、Ｃ＋＋言語では、クラスの定義のときに継承元を指定することによって実現される。例えば、インターネットも使えるテレビ（InternetTV）という新しいクラスを定義することを考えてみる。

そのとき、TelevisionクラスとInternetTVクラスは非常によく似ているので、Televisionクラスのうち、利用できるものは利用し、追加するものは追加することにより、新たにInternetTVクラスを定義することができる。

Ｃ＋＋言語では、継承元であるテレビクラスと継承先であるインターネットテレビクラスを、それぞれ「親クラスと子クラス」あるいは「基底クラスと導出クラス」と呼ぶ。
継承は、何代にも渡って行うことが可能である。継承の機能を利用して新たなクラスを定義する様子をリストⅢ－７に示す。

```
class InternetTV :public Television   // ①インターネットテレビクラス
    {
    protected:                        // ②外部には非公開の部分
        char  cInternetAddress[40];   //   ③インターネットのアドレス
    public:                           // ④公開の部分
        void  DisplayProgram(void);   //   ⑤番組を画面に表示する
        char* GetAddress(void);       //   ⑥インターネットアドレスを知る
        void  SetAddress(char* cText);//   ⑦インターネットアドレスを設定する
    };
```

リストⅢ－７　継承によるクラス定義の例（インターネットも使えるテレビ）

①の部分に、publicという予約語が記述されている。これは、publicな継承を使う意味である。また、ここに何も書かないか、privateという予約語を書くと、privateな継承という意味になる。

両者の違いは、クラスを継承したときに、３種類のアクセス属性がどのように継承されるかの違いである。両者の比較を表Ⅲ－2に示す。

表Ⅲ－2　継承の種類とアクセス属性

継承の種類	アクセス属性		
	private	protected	public
public継承	private	protected	public
private継承	private	private	private

もし、private継承を選んだ場合、親クラスのアクセス属性が何であっても、子クラスにおける親クラスの部分のアクセス属性はすべてprivateになってしまう。これは、子クラスから、更に継承によって新たなクラスを作ることができないことを意味する。すなわち、継承を用いたクラスの再利用が難しくなってしまう。

そこで、継承には、必ずpublic継承を使うようにするべきである。

次に、リストⅢ－８に継承後のクラスであるインターネットテレビクラスのメンバ関数の実際の処理部を示す。また、リストⅢ－９に継承後のクラスからオブジェクトを生成する例を示し、図Ⅲ－２にその実行結果を示す。

```
void  InternetTV::DisplayProgram(void)
    {                                   // 番組を画面に表示する
    if(0==GetPower())                   // 電源スイッチがオンの場合
        {
        if(0!=strcmp(GetAddress(),""))  // アドレスが設定されている場合
            {
            cout << "これはインターネットです。¥n";
            cout << "アドレスは[" << GetAddress() << "]です。¥n";
            cout << "音量は" << GetVolume() << "です。¥n";
            }
        else
            Television::DisplayProgram();  // 親クラスの同名のメンバ関数を呼ぶ
        }
    }
char* InternetTV::GetAddress(void)
    {
    if(0==GetPower())                   // 電源スイッチがオンの場合
        return cInternetAddress;        // インターネットアドレスを返す
    else                                // 電源スイッチがオフの場合
        return "";                      // ヌル文字列を返す
    }
void InternetTV::SetAddress(char* cText)
    {
    if(0==GetPower())                   // 電源スイッチがオンの場合
        strcpy(cInternetAddress,cText); // インターネットアドレスを設定する
    else                                // 電源スイッチがオフの場合
        strcpy(cInternetAddress,"");// ヌル文字列を設定する
    }
```

リストⅢ－８　メンバ関数の実際の処理部の例（インターネットテレビクラス）

```
void main(void)
    {
    InternetTV *ITV;        // ITVというオブジェクトへのポインタを用意する
    ITV=new InternetTV;     // ITVというオブジェクトを新たに確保する
    ITV->SwitchPower();     // 電源スイッチをオンにする
    ITV->DisplayProgram();// １チャンネルの番組を画面に表示する
    ITV->SetAddress("http://www.tokyo-u.ac.jp");
                            // 東京大学のホームページのインターネットアドレスを設定
    ITV->DisplayProgram();// 東京大学のホームページを画面に表示する
    delete ITV;             // TVというオブジェクトを解放する
    }
```

リストⅢ－９　継承後のクラスからのオブジェクトの生成例

```
テレビの電源がオンになりました。
これは1チャンネルです。
音量は5です。
これはインターネットです。
アドレスは[http://www.tokyo-u.ac.jp]です。
音量は5です。
```

図Ⅲ－２　リストⅢ－９の実行結果

(5) メッセージ

メッセージは、Ｃ＋＋言語においては、オブジェクトのメンバ関数を起動することで実現される。

メッセージの送受信は、送り手と受け手がいて初めて成立する。そこで、送り手と受け手との間で、送受信の方法についてあらかじめ取り決めておかなければならない。Ｃ＋＋言語においては、メッセージの送り手であるオブジェクト外部のプログラムから、受け手であるオブジェクト内部のメンバ関数の呼び出し方を取り決めておくことに相当する。

メッセージの受け手であるオブジェクトは、受け取ったメッセージによって依頼された操作を実行する。この場合、送り手は、オブジェクトの内部でどのようにメッセージが実現されているかということの詳細を知っている必要はない。送り手に要求されるのは、あくまでもメッセージを送ることである。

(6) 多様性

多様性は、Ｃ＋＋言語においては、異なるオブジェクトの同名のメンバ関数を起動することで実現される。例えば、既に定義したテレビ（Television）というクラスに加えて、エアコン（AirConditioner）及びステレオ（Stereo）という二つのクラスを新しく定義することを考えてみる。リストⅢ－１０に二つのクラスの定義部分を示す（メンバ関数の実際の処理部については省略する。）。

```
class AirConditioner                    // エアコンクラス
    {
    protected:                          // 外部には非公開の部分
        int    iPower;                  //    電源
    public:                             // 公開の部分
        void   SwitchPower(void);       //    電源スイッチをオン・オフする
    };

class Stereo                            // ステレオクラス
    {
    protected:                          // 外部には非公開の部分
        int    iPower;                  //    電源
    public:                             // 公開の部分
        void   SwitchPower(void);       //    電源スイッチをオン・オフする
    };
```

リストⅢ－１０　エアコンクラスとステレオクラスの定義部分

三つのクラスが揃ったところで、これらのクラスのSwitchPowerというそれぞれ同じ名前のメンバ関数を呼ぶプログラムを作成し、リストⅢ－１１に示す。また、その実行結果を図Ⅲ－３に示す。

```
void main(void)
    {
    Television *TV;         // TVというオブジェクトへのポインタを用意する
    AirConditioner *AC;     // ACというオブジェクトへのポインタを用意する
    Stereo *ST;             // STというオブジェクトへのポインタを用意する
    TV=new Television;      // TVというオブジェクトを新たに確保する
    AC=new AirConditioner;// ACというオブジェクトを新たに確保する
    ST=new Stereo;          // STというオブジェクトを新たに確保する
    TV->SwitchPower();      // TVの電源スイッチをオンにする
    AC->SwitchPower();      // ACの電源スイッチをオンにする
    ST->SwitchPower();      // STの電源スイッチをオンにする
    delete ST;              // STというオブジェクトを解放する
    delete AC;              // ACというオブジェクトを解放する
    delete TV;              // TVというオブジェクトを解放する
    }
```

リストⅢ－１１　三つのオブジェクトの多様性

```
テレビの電源がオンになりました。
エアコンの電源がオンになりました。
ステレオの電源がオンになりました。
```

図Ⅲ－３　リストⅢ－１１の実行結果

この実行結果から、異なる三つのクラスから生成したTV、AC、STという三つのオブジェクトに対して、SwitchPower、すなわち「電源スイッチをオン・オフせよ」という同じ内容のメッセージを送ったところ、三つのオブジェクトがそれぞれ別の動作をしたことがわかる。これがＣ＋＋言語で実現されている多様性である。

3　Ｃ＋＋言語でのオブジェクト指向のその他の事項の実現

ここでは、Ｃ＋＋言語において、オブジェクト指向の考え方がどのように実現されているかについて、前節で取り上げなかった事項について説明する。

(1) コンストラクタ・デストラクタ

「コンストラクタ」は、オブジェクトを生成するときに自動的に実行される初期処理のことである。この処理は、利用者からは見えないところで、あくまでも暗黙のうちに実行される。Ｃ＋＋言語においては、コンストラクタは、クラスと同じ名前のメンバ関数として定義する。

一方、「デストラクタ」は、オブジェクトを削除するときに自動的に実行される終了処理のことである。この処理も利用者からは見えないところで、あくまでも暗黙のうちに実行される。Ｃ＋＋言語においては、デストラクタは、クラスと同じ名前の頭に～を付けたメンバ関数として定義する。

これら二つの関数は、クラスを定義するときに記述しておけば、オブジェクト生成・削除時には暗黙のうちに実行されるため、オブジェクトの初期処理・終了処理の抜けがなくなり、ソフトウェアの品質を向上させることになる。

ただし、コンストラクタとデストラクタには、他のメンバ関数と異なり、戻り値を設定することはできない。そのため、メモリの確保のような実行時エラーが発生する可能性があるプログラムは、コンストラクタの中には記述しない方がよい。コンストラクタは、主にデータメンバに値を設定するために使用すべきである。

```
class Television                    // ①テレビクラス
    {
    protected:                      // ②外部には非公開の部分
                                    :
                                    :
    public:                         // ⑥公開の部分
        Television(void);           //    ⑦コンストラクタ
         Television(void);          //    ⑧デストラクタ
                                    :
    };
```

リストⅢ－１２　コンストラクタとデストラクタの定義の例

また、親クラスでコンストラクタやデストラクタが定義してある場合、子クラスからオブジェクトを生成・削除するときにも親クラスのコンストラクタ・デストラクタが呼び出されるので、そのことを意識してプログラムを作成する必要がある。

ここで、リストⅢ－３で示したテレビというクラスの定義の中の、⑦コンストラクタと⑧デストラクタの部分を、リストⅢ－１２としてもう一度示す。

(2) 関数プロトタイプ

Ｃ言語は、ある意味では非常にルーズな言語である。例えば、関数を定義したときの引数の型と、関数を実際に呼び出したときの引数の型が異なっている場合でも、純粋なＣ言語のコンパイラでは特にエラーにならず、それが究明の困難なバグの原因になることがある。そこで、Ｃ＋＋言語では、このようなバグの発生を防止するために、「関数プロトタイプ」という機能を用意している。

Ｃ＋＋言語における関数プロトタイプとは、全ての関数のインタフェースを実際の関数処理記述部分とは別に定義する必要があるというものである。このインタフェース定義のことを関数プロトタイプと呼ぶ。

関数プロトタイプを指定することにより、もし、関数プロトタイプと実際の関数呼び出しのインタフェースが一致していない場合、コンパイラがそれを検出し、エラー表示を行う。これによって、引数の型が異なっているなどのバグを未然に防止することができる。

また、最近では、Ｃ言語のプログラムにおいても、国際的な標準であるＡＮＳＩによって、プロトタイプを指定するという仕様が追加された。

リストⅢ－１３に関数プロトタイプと実際の関数定義の関係を示す。

```
[関数プロトタイプ]
void InternetTV::SetAddress(char* cText);

[実際の関数定義]
void InternetTV::SetAddress(char* cText)
    {
    if(0==GetPower())                    // 電源スイッチがオンの場合
        strcpy(cInternetAddress,cText); // インターネットアドレスを設定する
    else                                 // 電源スイッチがオフの場合
        strcpy(cInternetAddress,"");// ヌル文字列を設定する
    }
```

リストⅢ－１３　関数プロトタイプと実際の関数定義

(3) ヘッダファイル

Ｃ＋＋言語では、プログラム全体を１本にまとめても、クラスごとに分割してもコンパイルすることもできる。しかし、将来一部のクラスを再利用することを考えると、プログラム全体を１本にまとめるよりも、クラスごとにプログラムを分割してコンパイルしたほうが都合がよい。

分割コンパイルを行う場合、関数プロトタイプやクラス定義はヘッダファイルにまとめるようにする。

例えば、プログラム名がShowTV、クラス名がTelevisionとInternetTVであったとする。

その場合、

Televisionクラス定義	＝	TV.H
Televisionクラス実装部	＝	TV.CPP
InternetTVクラス定義	＝	ITV.H
InternetTVクラス実装部	＝	ITV.CPP
関数プロトタイプ	＝	SHOWTV.H
メインプログラムソース	＝	SHOWTV.CPP
ヘッダファイル	＝	COMMON.H

といったファイルにプログラムを分割する。

また、分割コンパイルを行う場合、ヘッダファイルを２度指定したことになることがある。もし、そのままコンパイルを行うと、「クラス定義が二重に行われた」などのエラーが発生する。これを避けるためには、マクロ命令で実際のヘッダを挟み込むことが最も効果的である。

リストⅢ－１４にCOMMON.Hというヘッダファイルの二重指定を防止するためのマクロ命令を示す。これは、__COMMON__Hという値が定義されているか否かによって、既にヘッダファイルが指定されているかを知るようになっている。

```
#if !defined(__COMMON__H)
#define       __COMMON__H
        (実際のヘッダ)
#endif    // __COMMON__H
```

リストⅢ－１４　ヘッダファイルの二重指定防止マクロ

(4) スコープ演算子

Ｃ＋＋言語におけるスコープ演算子とは、関数を呼び出すときに関数の前に::と記述し、プログラムの中のどの関数を呼び出したいのかを明示的に限定するものである。

例えば、テレビ（Television）クラスのメンバ関数SwitchPowerの実際の処理を記述した部分では、

```
void Television::SwitchPower()
```

と、メンバ関数の前にクラス名とスコープ演算子::を記述している。

また、次に説明する遮蔽定義でもスコープ演算子を使用している。更に、グローバルな変数と、クラス内で使用する変数の名前が同一であり、しかもグローバルな変数を使いたい場合にも、スコープ演算子を使用する。例えば、iNumberという名前がグローバル変数とクラス内の変数で重複した場合、

```
::iNumber
```

と記述することによって、明示的にグローバル変数iNumberが使用される。

(5) 遮蔽（しゃへい）定義

既にリストⅢ－７で示したようにメンバ関数DisplayProgramは、親クラスであるテレビクラスにも、子クラスであるインターネットテレビクラスにも存在する。このように親クラスと子クラスで同じ名前のメンバ関数が定義された場合、子クラスから生成したオブジ

ェクトでは、子クラスのメンバ関数の方がメッセージを優先して受け取り、親クラスのメンバ関数は隠されてしまう。このようなメンバ関数の定義の方法を、Ｃ＋＋言語では「遮蔽定義」あるいは「オーバライディング（overriding）」と呼んでいる。

しかし、子クラスのDisplayProgramというメンバ関数の中で、どうしても遮蔽定義され隠された親クラスのDisplayProgramというメンバ関数を呼びたいこともある。その場合は、リストⅢ－８にあるように、

```
Television::DisplayProgram()
```

と、メンバ関数の前に::（スコープ演算子）を記述し、その前に親クラス名を指定すればよい。

遮蔽定義は、このように親クラスのメンバ関数を子クラスで置き換え、改良するために使われることが多い。

(6) フレンド関数

Ｃ＋＋言語には、「フレンド関数」と呼ばれる機能がある。これは、クラスの中でfriend指示子を使って宣言された関数である。このフレンド関数は、このクラスのメンバ関数ではないが、クラスの中でprivateやprotected属性をもっている内部データの各変数に対して、自由に参照・更新することができる。

また、フレンド関数は、クラスのメンバ関数と異なり、受け手なしで実行することができるという特徴を持っている。

しかし、フレンド関数自体は、クラスの外部からクラスの内部データを自由に参照・更新するという、オブジェクト指向プログラミングの考え方に反するものである。フレンド関数をむやみに使用するのは、オブジェクト指向の見地上問題があるので、絶対必要な箇所以外は使用すべきではない。

(7) 参照

Ｃ＋＋言語では、「参照」という型をもつ変数を定義することができる。参照型をもつ変数は、値を一回設定した後は値を変えることができない。そして、参照型は、変数の前に＆（アンパサンド）を付けて宣言する。参照型の例をリストⅢ－１５に示す。また、実行結果を図Ⅲ－４に示す。

```
int   Clock;                    // Clockの時間を示す変数
int   &Watch=Clock;             // Clockへの参照
Clock=12;                       // Clockを１２時に設定
cout << "Clockの表示は" << Clock << "です。¥n";
cout << "Watchの表示は" << Watch << "です。¥n";
Watch++;                        // Watchを１時間進める
cout << "Clockの表示は" << Clock << "です。¥n";
cout << "Watchの表示は" << Watch << "です。¥n";
```

リストⅢ－１５　参照型の例

```
Clockの表示は12時です。
Watchの表示は12時です。
Clockの表示は13時です。
Watchの表示は13時です。
```

図Ⅲ－４　リストⅢ－１５の実行結果

参照が役に立つのは、参照型を関数の引数に指定した場合である。Ｃ言語は、ＦＯＲＴＲＡＮやＢＡＳＩＣといった言語と異なり、引数が値渡しである。そのため、関数の内部で引数に指定された変数の値を直接変更することはできない。もし値を変更したければ、引数をポインタ型にする必要がある。

しかし、ポインタ型で定義した変数は、＊（アスタリスク）を前に付けないといけない。一方、参照を使うと＊を付ける必要がない。

(8) 結合

Ｃ＋＋言語において「結合」とは、送り手から受け手に送られたメッセージが、特定のメンバ関数と結びついて起動されることをいう。

そして、結合には、それが起こる時期によって、静的結合と動的結合の２種類がある。静的結合は、プログラムのコンパイルあるいはリンクの時期に発生するものをいう。また、動的結合は、プログラムの実行時に発生するものをいう。静的結合は、性能的には良いが柔軟性に欠け、一方動的結合は、柔軟性は高いが性能的には良くないとされる。

Ｓｍａｌｌｔａｌｋは、すべて動的結合が行われる言語である。これは、プログラムの実行時にならないと結合が起こらないことを意味し、Ｓｍａｌｌｔａｌｋがインタプリタであるとされる所以である。

(9) 仮想関数

Ｃ＋＋言語においては、メンバ関数の定義にvirtualという予約語が使われていて、しかもその関数メッセージの受け手がポインタ値あるいは参照である場合に、動的結合が行われる。このようなとき、そのメンバ関数を「仮想関数」という。

```
class Ball
    {
    public:
        void Question(void);
        virtual void Answer(void);
    };

void Ball::Question(void)
    {
    cout << "これはボールですか？¥n";
    }

void Ball::Answer(void)
    {
    cout << "はい、ボールです。¥n";
    }

class Softball : public Ball
    {
    public:
        void Question(void);
        void Answer(void);
    };

void Softball::Question(void)
    {
    cout << "これはソフトボールですか？¥n";
    }

void Softball::Answer(void)
    {
    cout << "はい、ソフトボールです。¥n";
    }

void main(void)
    {
    Ball *pBall=new Ball;
    Softball *pSoftball=new Softball;
    pBall->Question();
    pBall->Answer();
    delete pBall;
    pBall=pSoftBall;
    pBall->Question();
    pBall->Answer();
    delete pSoftball;
    }
```

リストⅢ－１６ 仮想関数の例

```
これはボールですか？
はい、ボールです。
これはボールですか？
はい、ソフトボールです。
```

図Ⅲ－５　リストⅢ－１６の実行結果

Ｃ＋＋言語では、これ以外の場合には、すべて静的結合になる。オブジェクト指向言語の中には、Ｓｍａｌｌｔａｌｋのように、すべて動的結合が行われるものもあるのとは対照的である。仮想関数の例をリストⅢ－１６に示す。また、実行結果を図Ⅲ－５に示す。

関数Answerは、親クラスでvirtual付きの関数として定義されており、しかも関数メッセージの受け手が親クラスのポインタ値になっているので動的結合になる。一方、関数Questionは静的結合である。

(10) 多重定義

Ｃ＋＋言語では、引数の並びは違うが、名前は同じという関数を定義することができる。これは、一つの関数名に対して、二つ以上の関数本体が存在する場合があることを意味する。これにより、引数の並びやデータ型だけは異なるが、機能が同じ関数を複数作らなければならないとき、Ｃ言語のように、無理に関数名を変える必要がなくなった。

このような定義を「多重定義」と呼ぶ。そして、多重に定義された関数を区別するのには、引数の並びが一致しているか否かという情報を用いる。そのため、引数の並びのことを「引数署名」と呼ぶこともある。

多重定義の例をリストⅢ－１７に示す。これは、新年のメッセージを表示するPrintNewYear関数を多重定義したものである。引数がないとき、整数型の引数が１個だけあるとき、文字列型の引数が１個だけあるとき、整数型の変数が２個あるときで、新年のメッセージが変化する。また、その実行結果を図Ⅲ－６に示す。

```
void PrintNewYear(void)
    {
    cout << "あけましておめでとう。¥n";
    }

void PrintNewYear(int iOtoshidama)
    {
    cout << "お年玉は" << iOtoshidama << "円です。¥n";
    }

void PrintNewYear(char *cName)
    {
    cout << cName << "さん、新年あけましておめでとう。¥n");
    }

void PrintNewYear(int iMonth, int iDay)
    {
    cout << "今日は" << iMonth << "月" << iDay << "日です。¥n";
    }
          :
          :
void main(void)
    {
    PrintNewYear();
    PrintNewYear(10000);
    PrintNewYear("佐藤");
    PrintNewYear(1, 1);
    }
```

リストⅢ－１７　多重定義の例

```
あけましておめでとう。
お年玉は10000円です。
佐藤さん、新年あけましておめでとう。
今日は1月1日です。
```

図Ⅲ－６　リストⅢ－１７の実行結果

(11) 純粋仮想関数

Ｃ＋＋言語では、遮蔽定義を更に発展させたものとして、「純粋仮想関数」というものがある。具体的には、純粋仮想関数の定義の際に関数の名前の前にvirtualという予約語をつけ、関数の値を０に設定し、関数の実際の操作は記述しなければよい。純粋仮想関数は、関数の定義においてメンバ関数の名前を予約だけするものである。そして、これによって純粋仮想関数を有するクラスは、「仮想クラス」というものになる。

仮想クラスからは、オブジェクトを直接作成することはできない。もし、オブジェクトを作成しようとすると、コンパイルエラーが表示される。

純粋仮想関数の例をリストⅢ－１８に示す。これは、どのような人かを示すあいさつを表示するPrintGreeting関数が、親クラスであるPeopleクラスでは純粋仮想関数として定義されており、それが子クラスであるTeacherクラスとStudentクラスでは、PrintGreeting関数の実際の操作を記述している。また、その実行結果を図Ⅲ－７に示す。

```
class People                                    // 親クラス（仮想クラス）
    {
    virtual void PrintGreeting(void)=0;         // 純粋仮想関数
    };

class Teacher : public People                   // 子クラス
    {
    void PrintGreeting(void);
    };

class Student : public People                   // 子クラス
    {
    void PrintGreeting(void);
    };

void  Teacher::PrintGreeting(void)
    {
    cout << "こんにちは、わたしは教師です。¥n";
    }

void  Student::PrintGreeting(void)
    {
    cout << "こんにちは、わたしは学生です。¥n";
    }

void main(void)
    {
    Teacher teacher;
    Student student;
    teacher.PrintGreeting();
    student.PrintGreeting();
    }
```

リストⅢ－１８　純粋仮想関数の例

```
こんにちは、わたしは教師です。
こんにちは、わたしは学生です。
```

図Ⅲ－７　リストⅢ－１８の実行結果

(12) クラスライブラリ

オブジェクト指向プログラミング言語では、Ｃ＋＋言語に限らず、「クラスライブラリ」というものが用意されている。これは問題解決に必要であり、かつ、誰でもがよく使う汎用的なクラスをまとめたものである。大規模で複雑なソフトウェア開発を簡単に行うためには、このクラスライブラリを活用することが重要である。クラスライブラリで提供されているクラスは、既にテストがなされており、安心して使えるものである。しかも、クラスは、内部での実現方法を知らなくとも、使い方さえ知っていれば事足りる。そこで、ソフトウェアの開発者は、クラスライブラリのクラスを継承し、新たな機能を付け加えることによって生産性を向上させることができる。

Ｃ＋＋言語においても、有力なクラスライブラリがＣ＋＋コンパイラ開発メーカから提供されている。その代表例がマイクロソフト社のＭＦＣであり、ボーランド社のＯＷＬである。

4　Ｃ＋＋言語によるプログラミングの方針

Ｃ＋＋言語は、きわめて柔軟性が高い言語である。そのため、かなり注意深く使用しないと、オブジェクト指向プログラミングを行ったことにはならず、従来の手続き型プログラミングのままであることになりかねない。

Ｃ＋＋言語を使用する際には、オブジェクト指向プログラミングを行うという確固たる方針を立て、それに従ってプログラミングすることが重要である。

それでは、どのような方針を立てるべきなのであろうか。これまで提唱されてきた方針と、その理由を要約すると、次のようなものになる。

(1) グローバルな変数・関数の使用不可

グローバルな変数は、プログラムのどの場所からでも、いつでも変更される可能性がある。そのため、値が不正に変更されたといったバグが発生したときに、原因を特定することが大変困難になる。グローバルな関数も、プログラムのどの場所からでも、いつでも呼び出される可能性がある。そのため、同様の理由でバグの原因を特定することが困難になる。更に、グローバルな変数・関数を使うことによって、プログラムの各部分どうしの結合度が高まる。そのため、ある一部分だけを切り離して部品化することが困難になる。すなわち、プログラムの再利用の妨げになる。

そこで、グローバルな変数・関数は使用しないことである。

(2) データメンバへのprotected属性の指定奨励

データメンバにpublicなアクセス属性を指定した場合、クラスの外部からクラス内部の

データメンバを直接参照したり、変更されたりする可能性が生じる。その場合、グローバル変数と同じ状況になり、値が不正に変更されたといったバグが発生したときに、原因を特定することが大変困難になる。

一方、データメンバにprivateなアクセス属性を指定することは、継承により生成した子クラスからの参照を妨げることになり、プログラムの部品化という精神に逆行することになる。

そこで、データメンバには、protected属性を指定することを奨励する。

(3) フレンド関数の濫用不可

フレンド関数は、継承によって築いた親子クラスの関係を否定するものである。その場合、グローバル変数と同じ状況になり、値が不正に変更されたといったバグが発生したときに、やはり原因を特定することが大変困難になる。

そこで、フレンド関数の濫用は止め、必要不可欠な場合以外は使用しないことである。

(4) private継承の指定不可

継承を親、子、孫と同じように続けるためには、継承は、public継承でなければならない。すなわち、もしprivate継承にしてしまうと、継承を用いたクラスの再利用が大変困難になる。

そこで、private継承は使用しないことである。