3. C++言語

3.1. C++言語とは

C++言語とは，C 言語のスーパーセットとして作られたというよりも，オブジェクト指向のために C 言語をベースに新たに作った言語というのが正しいようです。しかし，実際には C 言語の機能はほとんど C++の中に取り込まれていますし，C の中で問題のあった機能は書き直され，より使いやすくなっています。新しい言語を覚えるというよりも，もっと便利になった C を使うという方が適切なようです。また，C++の特徴となっているクラスというものがあります。よく構造体と比較されますが(比較についてはここでは省略します。興味のあるひとは本でも見てみてください)，実際には構造体にはないとても便利な機能がたくさんあります。

C++でソースを書くには，これまでＣで拡張子を「c」にしていたのを「cpp」にするだけです。あとは特に気を付けることはありません。

3.2. C++の便利な機能

C++を使うことで，C と比較して便利になる機能を説明します。

3.2.1. デフォルト引数

C では，プロトタイプ宣言で記述された変数をきちんと書かないとエラーになりました。C++ではあらかじめデフォルトの引数を指定しておくと，引数の省略ができるようになります。

```
int func(double a = 1.2, double b = 3.4);
```

という具合です。これを呼び出すときは，

```
func();
func(5.6);
func(7.8.9.0);
```

と書きます。最初の例では，引数を全部省略しているのでfunc(1.2, 3.4)と同じ意味です。2番目の例では，引数を1つ省略しているのでfunc(5.6, 3.4)と同じ意味になります。最後の例は全部の値を指定しています。注意することは，最初の引数を省略して書くことはできないということです。

3.2.2. 変数の型宣言

C では，変数を関数内で宣言するときは必ず最初の方に書かないとエラーになりました。しかし，実際にソースを書いていくと途中で変数が欲しくなったりします。そういうときに，毎回先頭に戻って書くというのはとても不便です。しかし，C++は変数が必要になったとき，その変数が使われる前だったらどこで宣言してもいいことになっています。

ただし，間違いも起こりやすいので注意が必要です。list3.1 は間違いの例です。while 文の中で宣言した変数 a, n はループから抜けた時点で使えなくなってしまいます。この場合はコンパイラがエラーを出してくれるので気がつくはずですが，エラーの出ないような時は厄介なバグになります。

ちなみに，この例の場合，n の宣言に static を使っていますが，どうしてかわかりますか？ちょっと考えてみてください。

```
#include    <stdio.h>

int main(void)
{
    printf("数字を入力してください. ¥n");
    printf("0入力で終了します. ¥n");
    while(1){
        int a;
        static int n = 0;
        scanf("%d", &a);
        if(a == 0)
            break;
        n++;
        printf("第%d回の入力は%dです. ¥n",
            n, a);
    }
    printf("n = ", n);  // エラー！
    printf("a = ", a);  // エラー！
    return 0;
}
```

list3.1 変数宣言の間違いの例

3.2.3.　関数のオーバーロード

C++では関数のオーバーロードといい，引数が異なる同じ名前の関数を作ることができます。使い方によって，大変便利にも，厄介なバグの原因にもなります。使い方は，list3.2 の例を見れば一目瞭然でしょう。

```
#include    <stdio.h>

void print(int n);
void print(char *s);
void print(double d);

int main(void)
{
    print(1);
    print("Overload");
    print(123.4);
    return 1;
}

void print(int n){
    printf("%d¥n", n);
}

void print(char *s){
    printf("%s¥n", s);
}

void print(double d){
    printf("%lf¥n", d);
}
```

list3.2 関数のオーバーロード

3.2.4.　オペレータオーバーロード

オペレータとは「+, -, *, ... 」のことです。オペレータオーバーロードとはこれをオーバーロードするということです。関数のオーバーロードとほとんど同じです。一般的には後述のクラスと一緒に使いますが，クラス以外にも使いみちはあります。

オペレータオーバーロードの例としては，大抵の参考書が複素数の演算を出しています。ちょっと先走ってしまいますが，複素数クラスを使った例を示しましょう。ただし，list3.3 を見たらわかりますが，数字に double しか使えない，演算も「+」しかない簡略版です。

```
#include    <stdio.h>

class Complex{
    double real;    // 実数部
    double image;   // 虚数部
public:
    Complex(double r, double i);
    void print(void);
    Complex operator + (Complex x);
};

Complex::Complex(double r, double i){
    real = r;
    image = i;
}

void Complex::print(void){
    printf("%lf+%lfi¥n", real, image);
}

Complex Complex::operator + (Complex x){
    Complex temp(0.0, 0.0);
    temp.real = real + x.real;
    temp.image = image + x.image;
    return temp;
}

int main(void)
{
    Complex A(3.0, 2.0), B(5.0, 6.0);
    Complex C(0.0, 0.0);

    printf("A = ");
    A.print();
    printf("B = ");
    B.print();
    C = A + B;
    printf("A + B = ");
    C.print();
    return 1;
}
```

list3.3 オペレータオーバーロードの例

3.2.5.　new と delete

C 言語で配列の説明をしました。配列の宣言をするためにはあらかじめ配列のサイズを指定する必要がありました。しかし，実際に使うときにはプログラムを実行した後で，配列の大きさを決めたいときがあります。そのようなときに使うのがメモリの動的確保です。C 言語でも方法はありますが[1]，いくつか問題点もあるため，あまりお薦めできません。

1) 確保には「malloc」や「calloc」を，開放には「free」を使います．使用法が複雑なだけでなく，これらの関数にバグのある OS，コンパイラが多いので，できるだけ使わないようにとなっているものが多いです．

C++でメモリの動的確保を行うには new という演算子を使います。

new データ型;
new データ型[個数]

のように使います。失敗すると 0 を，成功すると領域の先頭アドレスを返します。使い終わったら，delete 演算子を使ってメモリの開放をします。list3.4 の例を見てみましょう。

```
#include    <stdio.h>

int main(void)
{
    int     n;
    printf("データの個数は？ : ");
    scanf("%d", &n);

    int *ptr;
    ptr = new int[n];
    for(int i = 0; i < n; i++){
        printf("data[%d] = ", i);
        scanf("%d", &ptr[i]);
    }

    printf("入力データの表示¥n");
    for(i = 0; i < n; i++){
        printf("data[%d] = %d¥n",
            i, ptr[i]);
    }
    delete []ptr;

    return 1;
}
```

list3.4 newとdeleteの例

確保したい領域のデータ型へのポインタを宣言し，ptr = new int[]; のように領域を確保します。使用が終わったら，delete [] ptr; のように領域を開放します。この場合のように，ptr が配列の場合は，空の[]を ptr の前に付けます。

3.3. クラス

C++の一番の特徴としてあげられるのがクラスです。一見するとクラスと構造体の違いは，関数が含まれるかどうかだけのように見えます。実際，使い方によっては全く違いがないときもあります。しかし，クラスにはオブジェクト指向を実現するための，アクセスコントロールとインヘリタンスという強力な機能を持っているという点で構造体と区別されます。

簡単なクラスの構造は，

```
class クラス名{
アクセスコントロール:
    クラスメンバ宣言;
};
```

です。アクセスコントロールとは，クラスメンバがメンバ以外からアクセスできるかどうかを制御します。アクセスコントロールの代表的なものに public と private があります。public メンバは外部からアクセスできるメンバ，private メンバは外部からアクセスできないメンバと考えてください。

```
class test{
    int a;
    char str[32];
public:
    int b;
    void ShowClass(void){
        . . . . . .
        . . . . . .
    }
};
```

＝

```
class test{
    int a;
    char str[32];
public:
    int b;
    void ShowClass(void);
};

void test::ShowClass(void){
    . . . . . .
}
```

左のように定義すると，a や str はプライベートメンバで外部からアクセスは不可能になり，b や関数 ShowClass はパブリックメンバで外部からアクセスが可能となります。また，関数 ShowClass の中身をそのまま書いていますが，これでは不便です。そこで，長い関数は右のように記述されることが多いです。ここで，「::」はスコープ演算子と呼ばれ。

戻値の型 クラス名 :: 関数名（引数，. . .）

のように使います。

実際に使った例を list3.5 に示します。

```
#include    <stdio.h>

class cl05{
    int                 a;
public:
    int                 b;
    void ShowClass(void);
};

void cl05::ShowClass(void){
    printf("数字を入力してください : ");
    scanf("%d", &a);
    printf("a = %d¥n", a);
    printf("b = %d¥n", b);
```

```
int main(void)
{
    cl05                first;
    cl05                second;

    first.b = 100;
    first.ShowClass();
    second.b = 200;
    second.ShowClass();

    return 1;
}
```

list3.5 クラスの例

3.3.1. コンストラクタとデストラクタ，メンバ関数

クラスの初期化と終了処理をする関数がコンストラクタとデストラクタです。これらには一般的な関数とは少し違った特徴があります。

1. コンストラクタは，クラスの名前と同じ名前である。
2. コンストラクタは，引数を取ることができる。オーバーロードもできる。
3. コンストラクタは，戻り値がない(void 型でもない)。
4. デストラクタは，クラスの名前の前にチルダ「~」を付ける。
5. デストラクタは，引数がない。戻り値もない。
6. オブジェクトの宣言と同時にコンストラクタが呼ばれ，スコープを失うと，デストラクタが呼び出される。
7. コンストラクタもデストラクタも省略されると暗黙のうちに引数や，中身のない関数が定義される。
8. コンストラクタもデストラクタも public な関数である。

ということです。では，list3.6 の例を見てみましょう。

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

class String{
private:
    int     len;
    char    *str;
public:
    String(char *string);
    ~String();
    void Print(void);
};

String::String(char *string){
    len = strlen(string);
    str = new char[len + 1];
    strcpy(str, string);
    printf("%sのコンストラクタ¥n", str);
```

```
String::~String(){
    printf("%sのデストラクタ¥n", str);
    len = 0;
    delete [] str;
}

void String::Print(void){
    printf("文字列は「%s」です. ¥n", str);
    printf("文字長は%d文字です. ¥n", len);
}

int main(void)
{
    String  Hitotsume("一つ目");
    String  Futatsume("二つ目");
    Hitotsume.Print();
    Futatsume.Print();
    return 1;
```

list3.6 コンストラクタ・デストラクタの例

ここでは文字列を扱う String というクラスを定義することとします。文字列を扱うために，文字列長 len と文字列を保持するための領域へのポインタ str を定義しておきます。コンストラクタ String では，文字列の長さを len に格納し，文字列を保持するための領域を new で確保してから，そこに引数で与えられた文字列を格納しています。デストラクタ ~String では，文字列の長さ len を 0 にして，コンストラクタで確保した領域を解放しています。コンストラクタとデストラク

タの中で表示しているのは，どこで呼ばれているかがわかるようにするためです。

次に，main 関数を見てみると，Hitotume と Futatsume というオブジェクト(クラスで作成した変数をオブジェクトと言います)を作成し，その後 Print 関数でそれぞれのオブジェクトを表示して，終了しています。

さて，このプログラムを実行するとどうなるでしょう。特にどこで，どのようにコンストラクタとデストラクタが呼ばれているかに注意して見てください。

メンバ関数については，中でもう既に使ってしまっています。String::Print がそうです。スコープ演算子がついている以外は，C で出てきた一般的な関数と同じです。

3.3.2.　インヘリタンス

インヘリタンス(inheritance)とは「相続」「継承」という意味の英語で，C++ではクラスを継承することができます。クラスの継承をするためには，クラスの定義で次のように書きます。

```
派生クラス ： アクセスコントロール 基本クラス {
    :
};
```

派生クラスとは，基本クラスを元にして新しく導出されるクラスのことをいいます。ここで注意しなくてはいけないのが，アクセスコントロールです。ここには public，protected，private のいずれかが入ります。このアクセスコントロールと基本クラスメンバのアクセスコントロールによって派生クラスからのアクセスが決定されます。まとめると表 3.1 のようになります。

継承先のクラスには基本クラスのメンバが全て含まれます。また，派生クラスで新たなメンバを追加することもできます。list3.7 のように宣言した基本クラス A，派生クラス B で，関数 ShowA()，関数 ShowB()，関数 main()からアクセスできる変数はどれかを確認してみましょう。

表3.1 インヘリタンスとアクセスコントロール

基本＼派生	public	protected	private
public	public	protected	private
protected	protected	protected	private
private	アクセス不可		

	a	b	c	d	e	f
ShowA()				―	―	―
ShowB()						
main()						

```
class A{
private:
    int a;
protected:
    int b;
private:
    int c;
    void ShowA(void);
};

class B : public A{
private:
    int d;
protected:
    int e;
public:
    int f;
    void ShowB(void);
};
```

```
void A::ShowA(void){
    printf("%d¥n", a);
    printf("%d¥n", b);
    printf("%d¥n", c);
}

void B::ShowB(void){
    printf("%d¥n", a);
        :
    printf("%d¥n", f);
}

int main(void)
{
    B test;
        :
    return 1;
}
```

list3.7 インヘリタンス

3.3.3.　関数のオーバーライド

基本クラスの関数を派生クラスで書き換えてしまうことを関数のオーバーライドと言います。手順は，

1. 基本クラスでオーバーライドされる関数を virtual として宣言する。(仮想関数という)
2. 継承クラスでオーバーライドする関数を宣言する。

です。例を list3.8 に示します。

3.3.4.　コンポジション

クラスはメンバとして(クラス)オブジェクトを持つことができます。この方法により新しいクラスを作ることをコンポジションといいます。オブジェクトを持つというだけではそれほど難しいことはありません。ただ，一つ注意しなくてはいけないのが，引数のあるコンストラクタをもつオブジェクトをもつコンポジションです。もし，基本クラス Kihon のコンストラクタが int

の引数を一つとるクラスの場合，

```
class Composition{
public:
    Kihon kihon(5);
};
```

と書くとどうなるでしょう？実際にソースを書いてみると判りますが，これはエラーになります。では，どうすればいいのでしょうか。このように引数を伴う場合は，

```
class Composition{
public:
    Kihon kihon;
};
```

と，引数なしで書いておき，コンストラクタで，

```
Composition :: Composition() : kihon(5){
    :
}
```

と，書きます。これをメンバイニシャライザといいます。

3.3.5.　多重継承

今までは継承する基本クラスは1つだけでした。基本クラスが複数ある場合はどうなるでしょう。2つのクラス class A と class B があるとき，これらを基本クラスとする派生クラス class C をつくるには，

```
class C : public A, public B {
    :
}
```

と書きます。A，B それぞれのクラスに同名の str というメンバ変数があるとき，C のなかでそれぞれにアクセスするには，A::str，B::str のようにスコープ解決演算子を使用します。

3.4.　ストリーム

3.4.1.　入出力ストリーム

C++でも printf や scanf のような C の関数は使えました。しかし，本来 C++で入出力するには cout (cerr)や cin という入出力ストリームというものを使うようにとなっています。これらはそれぞれ標準出力（標準エラー出力），標準入力を示すストリームで，

```
cout << a << "文字" << endl; // 画面に「変数 a」，「文字」，「改行」 を続けて表示する。
cin >> str;                  // キーボードからの入力を str という変数に格納する。
```

のように使います。また，この入出力ストリームを使うにはあらかじめ iostream.h を読み込んでおく必要があります。

さて，このストリームとは何でしょう。英語で stream とは「流れ」や「小川」という意味ですが，ここでは入出力の通り道を示していて，C++の中では ostream というクラスのオブジェクトとして記述されています。そして，中で使っている「<<」や「>>」はオペレータオーバーロードで定義された演算子です。

cin や cout は書式指定など余計なことをしなくても良い時にはとても便利です。

3.4.2.　ファイルストリーム

ファイルの入出力に関しても，cin や cout と同じようにストリームで管理することができます。ファイルに対するストリームは ofstream というクラスが，fstream.h の中で定義されています。

```
ofstream( );
ofstream( const char* szName, int nMode = ios::out, int nProt = filebuf::openprot );
ofstream( filedesc fd );
ofstream( filedesc fd, char* pch, int nLength );
```

コンストラクタがオーバーロードされています。最初のコンストラクタはオブジェクトを作成するだけです。2番目のコンス

トラクタは，出力するファイル名を引数にしています。残りの引数はデフォルト引数で省略すると，出力用にオープン(ios::out)，デフォルトモード(ios::openprot)となります。後のコンストラクタはヘルプなどで確認してください。

逆にファイルの入力には，ifstream というのを使います。大体想像ができますね。

【練習問題 3.1】 クラス，デフォルト引数、オペレータオーバーロード

list3.3 の複素数クラス Complex を完成させ，四則演算を行うことができるようにしなさい。コンストラクタにデフォルト引数を設定することで，実数部のみの複素数の指定もできるようになります。

【練習問題 3.2】 new と delete

練習問題 2.2 で作成したプログラムを，読み込むデータ行数によって配列サイズを変更できるように書き換えなさい。

1) ファイルを読み込み行数をカウントする。
2) 読込みに必要な配列を確保する。
3) 確保した配列にデータを読み込む。

【練習問題 3.3】

http://avse.bpe.agr.hokudai.ac.jp/~ici/pc/3/list8.zip

は，学生の氏名と英語，数学の点数を格納するクラスのサンプルです。Student.h，Student.cpp に CStudent クラスとその関数が定義されています。list8.cpp がメインとなっていますので，実際に実行して機能を確認してください。

また，StudentEx.h，StudentEx.cpp は CStudent クラスを基に化学の点数を格納できるようにした CStudentEx クラスのサンプルです。これは list8ex.cpp がメインとなっています。これらのソースから派生クラスの定義方法とその機能を理解してください。

【練習問題 3.4】

http://avse.bpe.agr.hokudai.ac.jp/~ici/pc/3/list9.zip

は，線形リストと呼ばれるデータ構造のサンプルです。List.h，List.cpp が基本クラス，list9.cpp がそのメインとなっています。また，これに先頭要素と末尾要素の検索機能を派生クラスを用いて実装した例を SrchList.h，SrchList.cpp に，メニューを追加したメインを list9s.cpp としていれてあります。これらの定義方法とその機能を理解してください。

[線形リスト]

線形リストとは，要素を一方向に連結したデータ構造です。各要素は次の要素を示すポインタを持っています。配列は異なり，データの移動を伴わずに挿入や削除を行えるというメリットがあります。練習問題 3.4 で示した名前を格納する線形リストは図で表すと図 3.1 のようになります。それぞれ示したデータとポインタを持つ各要素をノードと呼びます。

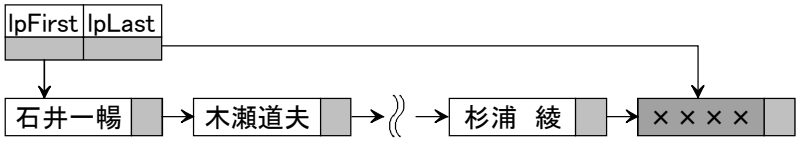

図 3.1　ダミーノードを用いた線形リストの例

線形リストを効率良く表現するために，先頭ノードと末尾ノードへのポインタを記憶する必要があります。また，末尾のノードはリストを管理するためのダミーのノードです。

コンストラクタが起動されるとダミーノード用の要素を1つ確保し，lpFirst と lpLast が共にその要素を示すように設定します。初期化された線形リストは図 3.2 のようになります。

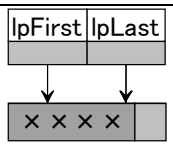

図 3.2　空の線形リスト

先頭への要素の挿入，末尾への要素の追加，先頭要素の削除，末尾要素の削除はそれぞれ図 3.3, 3.4, 3.5, 3.6 のようにポインタの変更と，新規要素の作成(new)または要素の削除(delete)で実装されます。

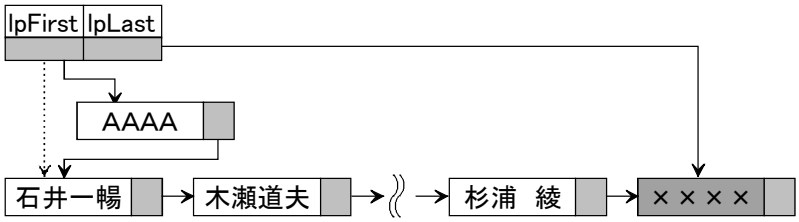

図 3.3　先頭への要素の挿入

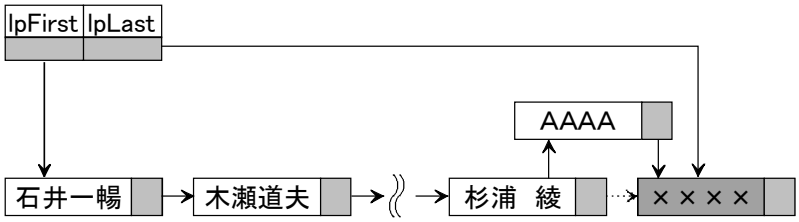

図 3.4　末尾への要素の追加

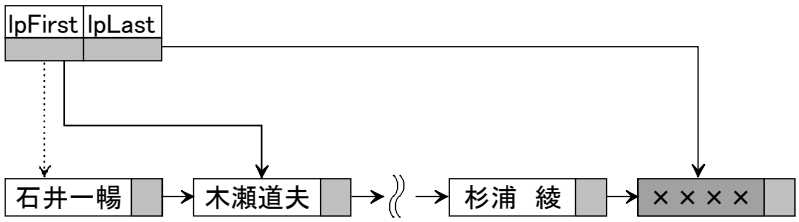

図 3.5　先頭要素の削除

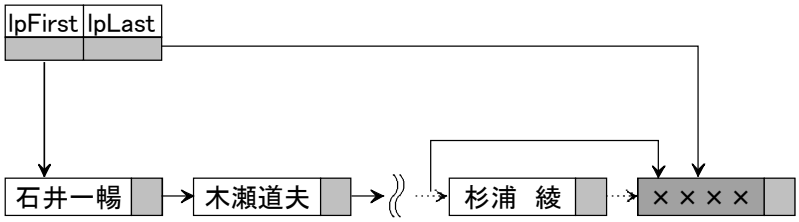

図 3.6　末尾要素の削除