オブジェクト指向言語C++ 前編　～C言語からC++へ～

筑波大学情報学群情報メディア創成学類 藤澤誠

1　C++とは

C++はC言語を拡張した言語です．一番の拡張はオブジェクト指向言語となったことによって,「クラス」が使えるようになったことです．もちろん,「クラス」以外にも便利な機能の追加や改良があります．これらのことを勉強していきます．ちなみにC++は「シー・プラス・プラス」や「シー・プラ・プラ」と呼んだりします．この輪講では読みやすさを重視して「シー・プラ・プラ」と呼ぶことにします．

では，これまでC言語でプログラムをしていたんだけど，C++でプログラムするときどうすればいいのか? 結論からいうと全部C言語で書いてもC++のプログラムになります．つまり，分からなくなったらC言語で書けばいいということです．違うのは，ファイルの拡張子が”*.c” から”*.cpp” に変わっただけです．試しに，自分の書いたC言語のプログラムの拡張子をcppに変えてみて下さい．VCでは何の問題もなくC++として処理されることでしょう．

2　参考文献

本輪講では，限られた時間でC++について述べていくので，C++のすべての機能を網羅することはできません．ある程度主要な機能のみ紹介するので，あとはここであげる参考文献をみる，購入するなどしてください．まず，C++ついての本としては

[1] 塚越一雄著,「決定版 はじめてのC++」, 技術評論社,1999

[2] 林晴比古著,「新C＋＋言語入門　ビギナー編」, ソフトバンクパブリッシング,2001

[3] 林晴比古著,「新C＋＋言語入門　シニア編 (上)(下)」, ソフトバンクパブリッシング,2001

また,

[4] B.W. カーニハン/D.M. リッチー著, 石田晴久訳,「プログラミング言語C」, 共立出版,1989

はC言語のバイブル的な存在なので是非買っておきたいところです.

また，Webページとしては

[5] 猫でもわかるプログラミング, http://kumei.ne.jp/c_lang/

は有名どころです.

3　Hello World

まず，最初はお約束のHello Worldから始めましょう．開発環境はVisual C++を想定しています．ついでに，入出力ストリームについてやります．Hello Worldプログラムの C言語版をList.1に，C++版をList.2に示しました.

3.1 標準入出力

まず，stdio.h のかわりに，iostream をインクルードします．List.2 の using namespace std はおまじないみたいなものと考えてください．

C 言語では，printf でデータを標準出力 (ここではディスプレイへの出力とします) へ渡します．このとき，%d や%f や%s などを使って，整数型や浮動小数点型や文字型といったものを指定します．間違って%d に浮動小数点型を渡したりすると，とんでもない出力となることを経験した人も多いと思います．

C++では，cout という命令で標準出力を行います．cout では，変数の型を気にすることなく出力できます．List.2 では，“Hello World!” という文字列型を出力しています．cout では，<<の右側に出力したい変数を書けば，型は勝手に判断してくれます．

```
float a = 1.2f;
int b = 3;
cout << "a = " << a << endl;
cout << "b = " << b << endl;
```

この出力は

```
a = 1.2
b = 3
```

となります．このように，変数なども型を気にせずにどんどん放り込んでやります．最後の endl は改行を示しています．

cout の他に標準エラー出力の cerr，標準入力の cin などがあります．

3.2 コメント文

C 言語ではコメント文に/* */を用いてきました．C++ではこれに加えて//が使えるようになっています．//は，//の左側から行末までをコメントとするものです．

List.1

```
#include <stdio.h>
int main(void)
{
  /* Hello の画面出力World */
  printf("Hello World!\n");
  return 0;
}
```

List.2

```
#include <iostream>
//名前空間の設定
using namespace std;
int main(void)
{
  // Hello の画面出力World
  cout << "Hello World!" << endl;
  return 0;
}
```

4 関数と変数

関数や変数についてもいくつかの追加，改良点があります．

4.1 変数の宣言

C 言語では，変数を関数内で宣言するとき、その関数の最初の方に宣言しないとエラーになりました．しかし，C++では，List.3 に示すように，その変数が使われる前ならば，関数のどこでも変数の宣言をすることができます．つまり，C++では必要となったときに変数を宣言すればよいということです．

しかし，これは注意が必要です．例えば，List.3 において，`cout << a << endl;` を `for` の後ろに出した場合，エラーとなります．つまり，ループ内で宣言された変数は，そのループ内でないと使えないということです．これを変数のスコープといいます．基本的には変数は `for` 文や`{ }`の中で宣言された場合はその中でしか使えません．

List.3

```
#include <iostream>
using namespace std;
int main(void)
{
  //変数の宣言
  for(int i = 0; i < 10; ++i){
    double a = (double)i*i;
    cout << a << endl;
  }
  return 0;
}
```

4.2 関数のデフォルト引数

C 言語では，関数を呼び出すとき，その関数で宣言されている引数をすべて指定しないとエラーになります．C++では，デフォルト値を設定しておくことで，いくつかの引数を省略することができます．List.4 にその例を示します．関数 `def_test` は，引数 `d` にデフォルト値 0 を設定しているため，`main` 側の呼び出しで `def_test()` と引数を指定しなかった場合は，`def_test(0)` と同じことになります．

関数のプロトタイプ宣言を用いる場合は，プロトタイプ宣言の方のみにデフォルト引数を設定します (両方に設定するとコンパイルエラーとなります)．

List.4

```
#include <iostream>
using namespace std;
//デフォルト引数
void def_test(int d = 0)
{
  cout << "d = " << d << endl;
}
int main(void)
{
  def_test();
  def_test(1);
  return 0;
}
```

また，次のようなデフォルト引数はエラーとなります.

```
void def_test(int d = 0, int e)
void def_test(int d, int e = 0, int f, int g = 0)
```

2 番目の例だと `def_test(1, 2, 3)` としたときに，e が省略されたのか，g が省略されたのかが分からないためです．そのため，デフォルト引数は，引数の宣言の最後にまとめる必要があります．この例では，

```
void def_test(int e, int d = 0)
void def_test(int d, int f, int e = 0, int g = 0)
```

としなければなりません.

4.3 関数のオーバロード

C++では，関数のオーバーロード (または関数の多重定義) ということができます．関数のオーバロードとは，引数の個数や種類が異なれば同じ関数名を付けてよいというものです．List.5 にその例を示します．同じ関数名 `func` である関数が 2 つあることに注意してください．呼び出し側では，指定する引数によってどの関数を呼び出すかを区別します.

先ほどのデフォルト引数を用いる場合は注意が必要です．例えば，

```
void func(void)
void func(int d = 0)
```

となると，どちらも `func()` となってしまうのでおかしくなります (まず，コンパイルエラーになるでしょうが).

List.5

```
#include <iostream>
using namespace std;
//関数オーバロード
int func(void)
{
  int d;
  cout << "Input␣Integer␣Number␣:␣";
  cin >> d;
  return d;
}
void func(int d)
{
  cout << d << endl;
}
int main(void)
{
  int d = func()
  func(d);
  return 0;
}
```

例えば，2 数を与えて，大きい方の値を返す関数 max を作りたいとします．関数のオーバロードはこういったときに役に立ちます．C 言語では，関数名が同じなのは認められていないため，変数型ごとに関数名を変えなくてはなりません．

```
int imax(int x, int y);
float fmax(float x, float y);
```

これは，あまり汎用的でないといえます．C++で関数のオーバロードを用いた場合，

```
int max(int x, int y);
float max(float x, float y);
```

とできます．ただし，引数に渡す数値の型は明示的に指定する必要があります (例えば，max(1.0f, (float)(2)) など)．

4.4 引数の参照渡し

C 言語や C++では，引数は基本的にコピー渡しとなります．このため，C 言語では，関数内部で実引数の値を変更したいときはポインタ引数を用います．List.6 は，ほとんどのプログラムの本に載っているスワップ関数です．List.6 の swap 関数内では，ポインタを引数に使ったため，逆参照演算子 (*) を使うことになり，なんだか見にくくなっています．

C++では，参照引数というポインタ引数を簡略化できる方法が用意されています．List.7 に swap 関数を参照引数で実現したものを示します．関数内部では，x,y を普通の変数のように扱え，呼び出し側も変数のアドレスを渡すということをしなくてよくなります．

List.6

```
#include <stdio.h>
void swap(int *a, int *b)
{
  int c;
  c = *a;
  *a = *b;
  *b = c;
}
int main(void)
{
  int a, b;
  printf("input 1 : ");
  scanf("%d", &a);
  printf("input 2 : ");
  scanf("%d", &b);

  /*結果の出力  */
  printf("input = (%d, %d)\n", a, b);
  swap(&a, &b);
  printf("output = (%d, %d)\n", a, b);
  return 0;
}
```

List.7

```
#include <iostream>
using namespace std;
void swap(int &a, int &b)
{
  int c;
  c = a;
  a = b;
  b = c;
}
int main(void)
{
  int a, b;
  cout << "input 1 : ";
  cin >> a;
  cout << "input 2 : ";
  cin >> b;

  //結果の出力
  cout << "input = (" << a << ", " << b << ")" << endl;
  swap(a, b);
  cout << "output = (" << a << ", " << b << ")" << endl;
  return 0;
}
```

5 new と delete

配列を使う場合，プログラムを書いている時点でどのくらいの大きさが必要なのかわからない場合がよくあります (例えば，あらかじめ数が分からないデータを読み込むとき，データの分布に応じた適応的構造を作るときなど)．十分な大きさの配列を用意しておくというのも一つの手ですが，確保した配列よりも大きなデータが入ってきた場合のエラー処理が必要になります．また，配列のために確保されたメモリ領域は，プログラム終了まで開放されないため，無駄となります．そのため，必要なときに，必要な量を確保するための仕組みが用意されています．これを動的確保といいます．

C 言語では，以下の malloc,calloc 関数を用います．それぞれの関数の定義は以下です．

```
void *malloc(size_t size)
void *calloc(size_t n, size_t size)
```

ここで，size_t は，確保する量を表すための符号なし整数型で，sizeof 関数の返値として得られます．これらの関数を使うためには，stdlib.h をインクルードする必要があります．

malloc 関数では，size の大きさのメモリ (配列) を確保します．calloc 関数では，size の大きさの領域を n 個確保します．両関数とも，確保したメモリへのポインタを返します．エラーの場合は NULL を返します．

実際に使うときは，このように使います．

```
int *p;
p = (int*)malloc(10*sizeof(int));
-------p を使った処理--------
free(p);
```

この例では，int 型で大きさ 10 の配列を確保しています．sizeof(int) は，int 型の大きさを返します．確保された配列へのポインタを int 型のポインタに変換し，p に渡します．確保した配列は，free 関数を用いて解放します．free しないと確保したメモリがプログラム終了後も残ってしまいます (これをメモリリークと呼ぶ)．そのため，free は，絶対に忘れてはいけない作業です．

C++では，malloc,free 関数の代わりに，new,delete 演算子を使います．

```
int *p;
p = new int[10];
-------p を使った処理--------
delete [] p;
```

確保された配列のサイズが 1 の時は単に delete p と書きます．new 演算子は，次回述べるクラスでも使えます．

5.1 2 次元配列の動的確保

前節で 1 次元配列を動的に確保する方法を述べました．では，2 次元配列の時はどうすればいいのでしょうか．一つの手としては，2 次元配列の内容を 1 次元配列に格納してしまうというのが考えられます．例えば，2 次元配列 x[h][w] を図 1 に示した矢印の順番で 1 次元配列 y[h*w] に格納するコードは以下となります．

```
for(int j = 0; j < h; ++j){
  for(int i = 0; i < w; ++i){
    y[j*w+i] = x[j][i];
  }
}
```

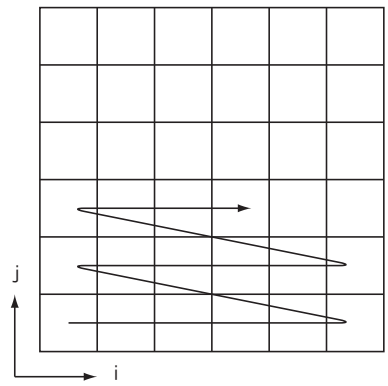

図 1: 2 次元配列を 1 次元配列に格納

また，2 次元配列を直接動的に確保する方法もあります．

```
double** x;
x = new double*[h];
for(int i = 0; i < h; ++i){
```

```
  x[i] = new double[w];
}
```

ここでは，double 型ポインタの配列 (要素数 h) 領域を確保し，その先頭アドレスを double** x に代入します．次に double 型データの配列 (要素数 w) 領域を h 個確保し、それらの先頭アドレスを x の各要素に順次代入するという操作をしています．メモリ領域の開放は上記の逆順で行います．

```
for(i = 0; i < h; ++i){
  delete [] x[i];
}
delete [] x;
```

6 オブジェクト指向とは

C++はオブジェクト指向言語 (OOL : Object Oriented Language) です．では，オブジェクト指向とは何でしょう．IT 用語辞典 e-Words より,「関連するデータの集合と，それに対する手続き (メソッド) を「オブジェクト」と呼ばれる一つのまとまりとして管理し，その組み合わせによってソフトウェアを構築する手法」だそうです．

身近なものでたとえるとわかりやすくなります．例えば，テレビを考えてください．テレビを「オブジェクト」と考えた場合，テレビ内にはそれを構成しているブラウン管や電子回路などのデータの集合とそれをうまく動作させるための手続きが格納されています．我々がテレビを利用するためには、例えばリモコンで適切なメッセージを与えるだけでよく，その内部構造まで知る必要はありません．このように個々の操作対象に対して固有の操作方法を設定することで，その内部動作の詳細を覆い隠し，利用しやすくしようとするものがオブジェクト指向です．

オブジェクト指向プログラミング (以下 OOP) は，プログラムで用いるデータとそれを操作する手続きをオブジェクトと呼ばれるひとまとまりの単位として一体化し、オブジェクトの組み合わせとしてプログラムを記述するプログラミング技法です．ただ，独立したオブジェクトが単に独立して動作するだけではなんのメリットもありません．独立しつつも複雑な連携を実行するが，それをユーザーに感じさせないというのが OOP で重要になります．そのための機能として，次の OOP の 3 つの柱と呼ばれる機能があります．

- カプセル化 (Encapsulation)
- 継承 (Inheritance)
- 多態性 (Polymorphism)

一般的に，この 3 つを備え持つプログラム言語を OOP とよびます．

7 構造体から始めよう

C++では**クラス**というものを使って前節で説明した機能を実現しています．ではクラスを説明するために，まず C 言語の構造体のおさらいから始めましょう．List.8 に構造体を使ったプログラムを示します．List.8 の構造体 Vec3 は 3 次元ベクトル (x, y, z) を格納するものです．main 文内では，この構造体用の変数を用意して，値を代入，画面出力しているだけです．

これをクラスで表現してみましょう．List.9 が単純にクラスにしたものです．struct の宣言を class に変えただけです．単純ですね．(OPP の機能を使ったとはいえませんが) これでクラスになりました．このときの変数 x,y,z を**メンバ変数**と呼びます．また，main 文内でクラス型の変数を定義しています (Vec3 v;)．この変

数 (v) を**オブジェクト**と呼びます．基本的には，このオブジェクトを使ってクラスのメンバ変数にアクセスします．

しかし，実はこのままコンパイルするとエラーになります．この理由を次章で考えていきましょう．

List.8

```
#include <iostream>
using namespace std;
typedef struct Vec3
{
  float x, y, z;
} Vec3;
int main(void)
{
  Vec3 v;
  v.x = 1;
  v.y = 2;
  v.z = 3;

  cout << "v␣=␣(" << v.x << ",␣";
  cout << v.y << ",␣";
  cout << v.z << ")" << endl;

  return 0;
}
```

List.9

```
#include <iostream>
using namespace std;
class Vec3
{
  float x, y, z;
};
int main(void)
{
  Vec3 v;
  v.x = 1;
  v.y = 2;
  v.z = 3;

  cout << "v␣=␣(" << v.x << ",␣";
  cout << v.y << ",␣";
  cout << v.z << ")" << endl;

  return 0;
}
```

8　アクセスコントロール

先の章で List.9 がコンパイルエラーになると言いましたが，そのエラー内容をみてみましょう．`main` 文内でいくつかでますが，すべて同じエラーです．(VC++.NET でコンパイル時のエラー)

```
private メンバ（クラス 'Vec3' で宣言されている）にアクセスできません。
```

ここで，**private** メンバというのがでてきました．これは，**アクセスコントロール**というもので，簡単に言えば，メンバ (メンバ変数とメンバ関数) が外からアクセスできるかどうかということを表したものです．private メンバは，外部からのアクセスができないというアクセスコントロールであるので，`main` 文から private メンバである `x,y,z` などにアクセスしようとしてエラーがでたのです．C++のクラスでは，何も指定しなければ private メンバになります．

実は構造体でもこういったアクセスコントロールがあります．では，先ほどの構造体ではなぜエラーにならなかったのでしょう．それは，構造体は何も指定しなければ **public** メンバと呼ばれるものになるからです．public メンバは，外からでも内からでも自由にアクセスできるメンバです．つまり，List.9 のコンパイルエラーを除くためにはメンバ変数 `x,y,z` を public メンバにすればよいのです．List.9 を以下のように変更してコンパイルしてみてください．

```
class Vec3
{
```

```
public:
  float x, y, z;
};
```

9 メンバ関数

クラスでは，変数だけでなく，関数をメンバにすることができます．この関数のことを**メンバ関数**といいます．List.9 に示したクラスにメンバ関数を追加してみましょう．追加するメンバ関数は，変数 `x,y,z` に値を代入する `input` 関数と画面出力する `output` 関数とします．List.10 にメンバ関数を追加したプログラムを示します．List.10 を実行したときの出力結果は，List.9 と同じになっていることを確認してみてください．

メンバ関数を追加する方法は大きく分けて 2 つあります．後で述べる**インライン関数**という形で追加する方法と，List.10 でやっている方法です．List.10 では，関数宣言 (プロトタイプ宣言) をクラス内に書き，クラス外にその実装を書いています．実装は，

返値 クラス名::メンバ関数名 (引数)

という形で書きます．

List.10

```
#include <iostream>
using namespace std;
class Vec3
{
public:
  float x, y, z;
  void input(float x0, float y0, float z0);
  void output(void);
};
void Vec3::input(float x0, float y0, float z0)
{
  x = x0; y = y0; z = z0;
}
void Vec3::output(void)
{
  cout << "v␣=␣(" << x << ",␣";
  cout << y << ",␣";
  cout << z << ")" << endl;
}
int main(void)
{
  Vec3 v;
  v.input(1.0f, 2.0f, 3.0f);
  v.output();
  return 0;
}
```

9.1 インライン関数

メンバ関数のもう一つの追加方法として，インライン関数があります．インライン関数では，コンパイル時に関数をインライン化 (関数の内容を呼び出し側に組み込む) します．これにより，関数呼び出しのオーバヘッドが無くなり高速化が期待できます．しかし，当然プログラムサイズは大きくなり，使用メモリ量も増えます．そのため，一般的に，インライン化は短い関数で，よく呼び出されるものに適用します (とはいえ最近のコンパイラは優秀なので最適化をオンにしておけば，勝手にインライン化してくれたりもします)．

List.11 に List.10 のクラス部分をインライン関数で書き直したものを示します．インライン化の仕方には，1. クラス内に直接実装を書く，2.`inline` 宣言する，の 2 種類があります

List.11

```
class Vec3
{
public:
  float x, y, z;
  void input(float x0, float y0, float z0);
  {
    x = x0; y = y0; z = z0;
  }
  void output(void);
};
inline void Vec3::output(void)
{
  cout << "v␣=␣(" << x << ",␣";
  cout << y << ",␣";
  cout << z << ")" << endl;
}
```

9.2 カプセル化

メンバ関数をただ追加するだけでなく，OOP の 3 つの柱の一つである**カプセル化**について考えてみましょう．1 章で内部の詳細を覆い隠し，利用しやすくしようとするものがオブジェクト指向であると述べました．これがまさにカプセル化ということです．

クラスのメンバ変数は外部から直接アクセスさせない，が基本です．List.10 では，入出力用のメンバ関数を用意したため，`main` 文から，直接はメンバ変数にアクセスしていません．そのため，メンバ変数 `x,y,z` を private メンバにします．また，メンバ関数 `output` はクラス内のメンバに変更を加えないので，こういう場合は，関数宣言の最後に `const` 識別子をつけます．`const` をつけると，その関数内でメンバを変更しようとするとコンパイル時にエラーがでるため，デバッグがしやすくなります．

```
class Vec3
{
private:
  float x, y, z;
public:
  void input(float x0, float y0, float z0);
  void output(void) const;
```

```
};
```

このようにすると，このクラスを使う側から内部のメンバ変数 x,y,z がどのように使われているかを覆い隠すことができます．使う方からすると単に入出力だけ考えていればよく，クラスの設計側から見ても内部の処理をどのように変更しても，input,output 関数だけ変えなければよいことになります．これがカプセル化です．

10 コンストラクタとデストラクタ

これまで作ってきたクラスでは，クラスのオブジェクトを生成した後に，input 関数を使って，値を代入し，出力すると言うことをやりました．C 言語では，例えば以下のように変数宣言時にその変数の初期化を行うことができたことを思い出してください．

```
int x = 1;
float y = 2.3f;
```

クラスでも初期化ができます．これを**コンストラクタ**と呼びます．また，後処理についても同様に用意されており，これを**デストラクタ**と呼びます．コンストラクタとデストラクタは，呼び出し側が明示的に呼び出さなくても，それぞれ自動的に，オブジェクトが生成されたときにコンストラクタ，オブジェクトを破棄するときにデストラクタが呼ばれます．

コンストラクタとデストラクタは特殊なメンバ関数としてクラスに記述します．List.12 に Vec3 クラスにコンストラクタとデストラクタを追加したプログラムを示します．コンストラクタとデストラクタはともに public メンバでなければなりません．また，返値はなく，関数名はコンストラクタがクラス名と同じ，デストラクタはクラス名の最初に ~(チルダ) をつけたものです．デストラクタは引数もとりません．

```
// コンストラクタ
クラス名 (引数)
// デストラクタ
~クラス名 ()
```

デストラクタは，1 つのクラスに 1 つだけしか存在しませんが，コンストラクタは複数存在できます．List.12 では，何も引数をとらないコンストラクタと引数を 3 つとるコンストラクタの 2 つを定義してあります．これは関数のオーバロードとなっていることに注意してください．オブジェクトを生成するときに引数を指定した場合は，Vec3(float x0, float y0, float z0) が呼ばれます．何も指定しなければ，Vec3() が呼ばれます．

List.12 では，コンストラクタとデストラクタ内で画面出力しているので，実際に実行してみて，どのタイミングでこれらが実行されるのかを確認してみてください．

10.1 デフォルトコンストラクタ

前節で述べたように，1 つのクラスに複数のコンストラクタを定義できます．一方，コンストラクタが 1 つも定義されていない場合でも，C++では裏で自動的にコンストラクタが作られます．これを**デフォルトコンストラクタ**と呼びます．

デフォルトコンストラクタは，引数がなく関数本体も空のコンストラクタです．

```
Vec3::Vec3()
{
}
```

これは，全く意味のない関数のように思えますが，オブジェクトが作成されるとき，プログラマからは見えない裏の初期化が必要とされるため，どのようなクラスでもコンストラクタがある必要があります．デフォルトコンストラクタは，コンストラクタを持たないクラスでも必ずコンストラクタを実行させるために作られます．

コンストラクタにはこのほかにもコピーコンストラクタと呼ばれるコンストラクタ (というよりはプログラム手法ですが) があります．

List.12

```
class Vec3
{
private:
  float x, y, z;
public:
  //コンストラクタとデストラクタ
  Vec3();
  Vec3(float x0, float y0, float z0);
  ~Vec3();
public:
  void input(float x0, float y0, float z0);
  void output(void);
};
Vec3::Vec3()
{
  cout << "constructor" << endl;
}
Vec3::Vec3(float x0, float y0, float z0)
{
  x = x0; y = y0; z = z0;
  cout << "constructor2" << endl;
}
Vec3::~Vec3()
{
  cout << "destructor" << endl;
}
int main(void)
{
  Vec3 v(1.0f, 2.0f, 3.0f);
  v.output();
  return 0;
}
```

11　インスタンス

C++では new,delete 演算子でメモリの動的確保を行うことを述べました．そのとき，new 演算子ではクラスでも使えると言いました．もちろん，クラスのメンバ変数にポインタを持たせておき，コンストラクタで

new して，デストラクタで delete するという使い方はよく使われます．この他に，クラスのオブジェクトを new で生成することもできます．List.13 にその例を示します．このように new で確保されたオブジェクトをインスタンスと呼ぶことが多いようです．インスタンスは，オブジェクトとほぼ同義語のように用いられることが多いのですが，実際にメモリ上に配置されたデータの集合という意味合いが強くなっています．

List.13

```
int main(void)
{
  Vec3 *vp;
  vp = new Vec3(1.0f, 2.0f, 3.0f);
  vp->output();
  delete vp;
  return 0;
}
```