第四章 陣列、指標與參考

4-1 物件陣列

4-2 使用物件指標

4-3 this指標

4-4 new 與 delete

4-5 更多有關new與delete

4-6 參考(reference)

4-7 傳遞物件參考

4-8 傳回參考

4-9 獨立參考與其限制

4-1物件陣列: 範例一

```
class samp {
    int a ;
public:
    samp() { a = 0 ; }
    int get_a() { return a ;}
} ;
void main() {
    samp ob[4] ; // 生成了?個ob物件
}
```

範例二：會呼叫甚麼建構子?

```
class samp {
    int a ;
public:
    samp(int n) { a = n ; }
    int get_a() { return a ;}
} ;
void main() {
    samp ob[4] = {1, 2, 3, 4} ;
    cout << ob[0].get_a() <<endl ;
}
```

範例三：多維陣列的初值

```
void main() {
    samp ob[4][2] = {1, 2, 3, 4,5,6,7,8} ;
    // samp ob[4][2] = {{1, 2}, {3,
    4},{5,6},{7,8}} ;
    cout << ob[3][0].get_a() <<endl ;
}
```

4-3 this指標

```
class frac {
    int q, p ;
public:
    frac() { this->q = 0; this->p = 0 ; }
    void set(int u, int d) {this->q = u; this->p
    =d ; }
} ;
void main() {
    frac f ;
    f.set(3, 5) ; // 傳了三個參數&f, 3, 5
}
```

4-4, 4-5 new 與 delete

- 動態記憶體配置
  - frac *f1 = new frac ;
  - frac *f2 = new frac() ;
  - frac *f3 = new frac(5, 3) ;
  - frac *f4 = new frac(f3) ; // f1 is an frac object
  - frac *f5 = new frac[20] ;
  - delete f1; delete[] f2; ……
  - delete[] f5;

4-4, 4-5 new 與 delete

- 以下各行指令，何時有frac物件產生?

```
1  frac f ;
2  frac *pf ;
3  pf = new frac() ;
4  frac fs[20] ;
5  frac *pfs ;
6  pfs = new frac[20] ;
```

4-6 參考型態(reference type)

- 爲變數建立別名(alias)

```
int x = 60 ;
int& LB = x ; // LB的資料型態是?
LB++ ; cout << x ;
// 注意: 沒有 *
frac f(3,5) ;  frac& pratio = f ;
pratio.set(7,10); f.show() ;
const int &ref = 10 ;
```

參考型態(reference type)

```
int ival = 1024;
int &refval = ival;
 // ok: refval is a reference to ival
int &refval2;
// error: reference must be initialized to an object
int ival = 1024;
int &refval = &ival;
// error: refval is of type int, not int*
int *p = &ival
// ok: refptr is a reference to a pointer
int *&refptr = p;
```

參考型態(reference type)

```
main(){
int ival = 1024, ival2
    = 2048;
int *p = &ival, *p2 =
   &ival2;
  cout<<*p<<" "
    <<*p2<<endl;
p=p2;
  cout<<*p<<" "
    <<*p2<<endl;
 }
```

```
main(){
  int ival = 1024, ival2 =
2048;
  int &ri = ival, int &ri2 =
ival2;
  cout<< ri<<"
"<<ri2<<endl;
  ri=ri2;
   cout<< ri<<"
"<<ri2<<endl;
 }
```

參考型態(reference type)

```
class frac {
    int q, p ;
public:
    frac() { this->q = 0; this->p = 0 ; }
    frac(int a, int b){q=a;p=b;}
    void set(int u, int d) {this->q = u; this->p =d ; }
   void show(){cout<<p<<" "<<q<<endl;}
} ;
void main(){
    frac f(3,5) ;
   f.show() ;
    frac& pratio = f ;
   pratio.set(7,10);
    f.show() ;}
```

4-6 參考型態(reference type)

- 使用時機
  - 獨立參考
  - 函數參數傳遞
  - 函數回傳值

範例二:

```
#include <math.h>
void round1(double num) ;  // void
    round1(double);
void round2(double &num) ; // void
    round2(double&);
void main() {
    double i = 100.4 ;
    round1(i) ;     cout << i ;
    round2(i) ;     cout << i ;
}
void round1(double num) {
    num = num/fabs(num) * int(fabs(num)+0.5) ;
}void round2(double &num) {/*四捨五入 同上
    */ ....}
```

範例一: swap(x,y)

```
void swap(int &x, int &y) ; //如果無&?
void main() {
    int i =10, j =19 ;
    cout << i << " " << j <<endl ;
    swap(i, j) ; //注意呼叫方式，非swap(&i, &j);
    cout << i << " " << j << endl ;
}
void swap(int &x, int &y) {
    int temp = x; x = y ; y = temp ;
}
```

4-7 傳遞物件參考

```
class myclass {
    int data[20] ;
public:
    myclass() { ..... }
    ......
} ;
void fun1(myclass ob1) {..... } // 被呼叫時是否產生物件
void fun2(myclass& ob2) {.....} // 被呼叫時是否產生物件
void main() {
    myclass ob;  fun1(ob) ; fun2(ob) ;
}
```

Q: fun1()與fun2()哪個較有效率? 較省空間?

4-8 傳回參考

- 範例一: 寫在等式右邊函式

```
int score[50] ;
int& last() { return score[49]; }
void main() {
    last() = 0 ; // score[49] = 0 ;
    cout<<last()<<endl;
}
```

範例二:
不要傳回超過作用範圍的物件

```
// 以下程式有何錯誤?
int& last() { int a[50]; return a[49]; }
void main() {
    last() = 0 ;
    cout<<last()<<endl;
}
```

範例三: safeArray (務必讀懂)

```
class array {
    char *p; int size; ……
}
void main() {
    array a[20] ;
    //自動檢查錯誤
    // 如果你已overload []這個運算符號
    a[2] = 10; a[30] = 7;
}
```