國立屏東大學 資訊工程學系 物件導向程式設計

20. 運算子重載

重載(overloading)是指同樣名稱的成員函式可以擁有多個版本，而運算子重載，則是允許我們對運算元定義自己的運算子版本。

20.1 可重載的運算子

可重載的運算子如表table 1:

+	-	*	/	%	^	&	|	~	!	,	=
<	>	< =	> =	++	- -	< <	> >	==	!=	&&	||
+=	-=	/=	%=	^=	&=	|=	*=	< < =	> > =	[]	()
- >	- >*	new	new[]	delete	delete[]						

Tab. 1: 可重載的運算子列表

table 2則是不可被重載的運算子:

: :	. *	.	?:

Tab. 2: 不能被重載的運算子列表

20.2 運算子重載的語法與語意規則

- C++語言規定內建資料型態的運算子是不可以被重載的，因此，在重載運算子時，相關的運算元至少要有一個是自定的型態或類別。
- 運算子的優先順序、結合律(左關聯或右關聯)是固定不變的。

運算子重載是以函式宣告的方式來進行，其語法如下：

```
ReturnType operatorOP (type op1[, type op2]?);
```

其中OP是要重載的運算元，op1與op2則是運算元，ReturnType是計算後傳回值的型態。下面是一個例子：

```
#include <iostream>
using namespace std;

struct Point
{
  int x;
  int y;
```

```
};


Point operator+(Point p1, Point p2)
{
  Point p;
  p.x = p1.x + p2.x;
  p.y = p1.y + p2.y;
  return p;
}

int main()
{
  Point px={5,6}, py={7,8};
  Point pz;

  pz = px+py;

  cout << "(" << pz.x << "," << pz.y << ")\n";
  return 0;
}
```

在上例中，我們設計了一個加法的運算子重載，用以處理Point + Point這種運算，其中p1與p2為運算元，傳回值則為一個Point類別的物件。為了要減少「傳值(call by value)所需的複製」，也可以改成call by reference

```
Point  operator+(Point &p1, Point &p2)
{
  Point p;
  p.x = p1.x + p2.x;
  p.y = p1.y + p2.y;
  return p;
}
```

通常，若運算子本身只是以其值進行運算，並不會在函式中改變其值，因此，還可以改寫如下：

```
Point  operator+(const Point &p1, const Point &p2)
{
  Point p;
  p.x = p1.x + p2.x;
  p.y = p1.y + p2.y;
  return p;
}
```

若是要進行例如「++」或「+=」這一類的運算時，其運算的傳回值為運算元本身，例如x+=5其意涵為x = x + 5我們必須以x+5的值做為x的值，所以x不但是傳入的參數(運算元之一)，同時也是傳回值

所要存放的地方。請參考下面的程式碼：

```
Point & operator+=(Point &p1, const Point &p2)
{
  p1.x+=p2.x;
  p1.y+=p2.y;
  return p1;
}
```

看了幾個例子後，現在讓我們試著重載「< <」好讓輸出變得更容易。換言之，我們可以使用cout « p1;這種方式來輸出。注意到這個敘述，其中「< <」為運算子，而cout與「p1則為運算元。下面是一個重載的例子：

```
void operator<<(ostream &out, Point &p)
{
  out << "(" << p.x << "," << p.y << ")";
}
```

注意：一般我們用以輸出的cout是ostream類別的物件。

執行看看cout < < p1;與「cout < < p1 < < endl;看看有什麼差別？

為了要讓後續其它的「< <」也能正確地處理，我們將其改為：

```
ostream & operator<<(ostream &out, Point &p)
{
  out << "(" << p.x << "," << p.y << ")";
  return out;
}
```

如此一來cout < < p1 < < endl;就會先執行cout < < p1並傳回一個cout再進行cout < < endl;

20.3 類別的運算子重載

現在讓我們將上面的例子，改以類別方式實作，

```
#include <iostream>
using namespace std;

class Point
```

```
{
public:
  int x;
  int y;
};

ostream & operator<<(ostream &out, Point &p)
{
  out << "(" << p.x << "," << p.y << ")";
  return out;
}

Point & operator+=(Point &p1, const Point &p2)
{
  p1.x+=p2.x;
  p1.y+=p2.y;
  return p1;
}

Point operator+(const Point &p1, const Point &p2)
{
  Point p;
  p.x = p1.x + p2.x;
  p.y = p1.y + p2.y;
  return p;
}

int main()
{
  Point px={5,6}, py={7,8};
  Point pz;

  pz = px+py;

  cout << "(" << pz.x << "," << pz.y << ")\n";
  pz+=px;

  cout << "(" << pz.x << "," << pz.y << ")\n";
  cout << pz << endl;

  return 0;
}
```

有沒有發現，其實除了將 struct 改成 class 外，並沒有其它的修改。這是因為原本所設計的這些運算子重載函式都是以一般的函式來實作，但類別可以改成以成員函式方式實作：

```
#include <iostream>
using namespace std;
```

```
class Point
{
public:
  int x;
  int y;

  Point & operator+=(const Point &p)
  {
    x+=p.x;
    y+=p.y;
    return *this;
  }

  Point operator+(const Point &p)
  {
    Point newp;
    newp.x = x + p.x;
    newp.y = y + p.y;
    return newp;
  }
};

int main()
{
  Point px={5,6}, py={7,8};
  Point pz;

  pz = px+py;

  cout << "(" << pz.x << "," << pz.y << ")\n";

  pz+=px;

  cout << "(" << pz.x << "," << pz.y << ")\n";

  return 0;
}
```

要注意的是，當以成員函式實作運算子重載時，其參數部份已隱含了一個「看不見的參數」，也就是this指標，它會做為「隱形的」第一個參數，所以上述的程式中，其運算子重載的函式之參數都較結構體版本的少了一個參數。

但是，如果我們要實作「<<」的重載，若this做為第一個參數，這就會帶來問題，例如：

```
ostream & operator<<(ostream &out)
  {
    out << "(" << x << "," << y << ")";
    return out;
```

```
  }
```

由於this是第一個參數，out是第二個參數，當我們執行cout « p1;時，就會遇到型態不相符的問題，事實上這樣的重載，其支援的使用方式為p1 « cout;，這似乎沒有意義。因此，我們必須將之改以非成員函式的方式來實作：

```
#include <iostream>
using namespace std;

class Point
{
public:
  int x;
  int y;

  Point & operator+=(const Point &p)
  {
    x+=p.x;
    y+=p.y;
    return *this;
  }

  Point operator+(const Point &p)
  {
    Point newp;
    newp.x = x + p.x;
    newp.y = y + p.y;
    return newp;
  }
};


ostream & operator<<(ostream &out, Point &p)
{
  out << "(" << p.x << "," << p.y << ")";
  return out;
}


int main()
{
  Point px={5,6}, py={7,8};
  Point pz;

  pz = px+py;
  cout << "(" << pz.x << "," << pz.y << ")\n";

  pz+=px;

  cout << "(" << pz.x << "," << pz.y << ")\n";
```

```
  cout << pz << endl;

  return 0;
}
```

下面則是關於「>>」的實作:

```
istream & operator>>(istream &in, Point &p)
{
  in >> p.x >> p.y ;
  if(!in)
    p.x=p.y=0;
  return in;
}
```

20.4 前置與後置運算子重載

假設要重載「++」這樣的運算子，還有一個問題必須處理，那就是如何區分前置與後置？例如□□i++□ 與「++i□的差異。關於此點□C++使用不同的函式原型來區分:

```
//前置

Point & operator++(Point &p)
{
  p.x++;
  p.y++;
  return p;
}

//後置
Point & operator++(Point &p, int)
{
  p.x++;
  p.y++;
  return p;
}
```

20.5 問題討論

以point4.cpp為例

```
#include <iostream>
using namespace std;

class Point
{
public:
  int x;
  int y;

  Point & operator+=(const Point &p)
  {
    x+=p.x;
    y+=p.y;
    return *this;
  }

  Point operator+(const Point &p)
  {
    Point newp;
    newp.x = x + p.x;
    newp.y = y + p.y;
    return newp;
  }
};


istream & operator>>(istream &in, Point &p)
{
  in >> p.x >> p.y ;
  if(!in)
    p.x=p.y=0;
  return in;
}

ostream & operator<<(ostream &out, Point &p)
{
  out << "(" << p.x << "," << p.y << ")";
  return out;
}


int main()
{
  Point px={5,6}, py={7,8};
  Point pz;

  pz = px+py;

  cout << "(" << pz.x << "," << pz.y << ")\n";

  pz+=px;
```

```
  cout << "(" << pz.x << "," << pz.y << ")\n";
  cout << pz << endl;

  cin >> pz;
  cout << pz;

  return 0;
}
```

如果我們在main()函式中，使用cout << px + py << endl;那麼在編譯時就會遇到錯誤。其原因在於「+」與「<<」這兩個運算子重載必須一致，例如，以下的兩種方式，都可以解決上述的問題。

```
 Point operator+(const Point &p)
  {
    Point newp;
    newp.x = x + p.x;
    newp.y = y + p.y;
    return newp;
  };

ostream & operator<<(ostream &out, Point p)
{
  out << "(" << p.x << "," << p.y << ")";
  return out;
}
```

或是

```
Point & operator+(const Point &p)
  {
    Point *newp=new Point;
    newp->x = x + p.x;
    newp->y = y + p.y;
    return *newp;
  };

ostream & operator<<(ostream &out, Point &p)
{
  out << "(" << p.x << "," << p.y << ")";
  return out;
}
```

```
#include <iostream>
using namespace std;

class Point
{
public:
  int x;
  int y;

  Point & operator+=(const Point &p)
  {
    x+=p.x;
    y+=p.y;
    return *this;
  };

  Point & operator+(const Point &p)
  {
    Point *newp=new Point;
    newp->x = x + p.x;
    newp->y = y + p.y;
    return *newp;
  };
};


istream & operator>>(istream &in, Point &p)
{
  in >> p.x >> p.y ;
  if(!in)
    p.x=p.y=0;
  return in;
}
/*
void operator<<(ostream &out, Point p)
{
  out << "("<< p.x << "," << p.y <<  ")";
}
*/
ostream & operator<<(ostream &out, Point p)
{
  out << "(" << p.x << "," << p.y << ")";
  return out;
}
int main()
{
  Point px={5,6}, py={7,8};
  Point pz;


  pz = px+py;
```

```
  cout << "(" << pz.x << "," << pz.y << ")\n";

  pz+=px;

  cout << "(" << pz.x << "," << pz.y << ")\n";

  cout << pz << endl;

  cout << "----" << endl;
  cin >> pz;

  cout << pz+px;
  return 0;
}
```

20.6 friend函式

本章為了方便討論起見，將Point類別的資料成員皆暫時宣告為□public□□現在讓我們將其改回□private□□請參考下面的片段：

```
class Point
{
private:
  int x;
  int y;
...
```

編譯後發現許多錯誤，分別是：

- 因為資料成員變成私有的(private)□所以物件初始值的給定不能再用「={}」，必須要改成用建構函式進行。
- 在main函式中有些使用到x或y的程式碼，必須改掉
- 但仍然在「< <」與「> >」的重載上遇到存取私有資料成員的問題

前面已經討論過，這兩個重載函式必須定義成為非成員函式(也就是一般函式)，但如此一來，就讓其無法使用Point類別的私有資料成員。下面的方法是在Point類別的宣告中，將這兩個重載函式定義為Point之友，那就可以讓它們存取其私有資料成員了：

```
  friend istream & operator>>(istream &in, Point &p);
  friend ostream & operator<<(ostream &out, Point p);
```

```
#include <iostream>
using namespace std;

class Point
{
private:
  int x;
  int y;

public:
  Point(){};
  Point(int x, int y)
  {
    this->x=x;
    this->y=y;
  };

  Point & operator+=(const Point &p)
  {
    x+=p.x;
    y+=p.y;
    return *this;
  };
  Point & operator+(const Point &p)
  {
    Point *newp=new Point;
    newp->x = x + p.x;
    newp->y = y + p.y;
    return *newp;
  };
/*
  Point operator+(const Point &p)
  {
    Point newp;
    newp.x = x + p.x;
    newp.y = y + p.y;
    return newp;
  };
*/
  friend istream & operator>>(istream &in, Point &p);
  friend ostream & operator<<(ostream &out, Point p);
};


istream & operator>>(istream &in, Point &p)
{
  in >> p.x >> p.y ;
  if(!in)
    p.x=p.y=0;
  return in;
}
```

```
ostream & operator<<(ostream &out, Point p)
{
  out << "(" << p.x << "," << p.y << ")";
  return out;
}


int main()
{

  Point px(5,6), py(7,8);
  Point pz;

  pz = px+py;

  cout << pz << endl;

  pz+=px;

  cout << pz << endl;

  cin >> pz;
  cout << pz+px;

  return 0;
}
```