Programación en C++ (6) : sobrecargo de operadores

Dr. J.B. Hayet

CENTRO DE INVESTIGACIÓN EN MATEMÁTICAS

Septiembre 2009

Outline

Outline

Operadores

Son nada más que funciones, con una sintaxis un poco particular, por ejemplo :

```
double a=1.0,b=10.0;
double c=a+b;
```

El término "a+b" se puede leer también como el resultador del operador "+" que ha tomado dos argumentos, a y b. El C++ permite sobrecargar (*overload*) esos operadores para clases definidas por el usuario, por ejemplo para poder escribir :

```
Image a,b,c;
cout << a;
c = a + b;
c+= a;
```

Operadores : sintaxis

```
class Image {
public:
  int width;
  int height;
  Image(int w,int h) {
    width=w; height=h;
  };
  Image operator+(const Image &otra) {
    int w  = (width>otra.width)?width:otra.width;
    int h  = height + otra.height;
    return Image(w,h);
  }
};
```

Operadores : sintaxis

Un operador definido dentro de una clase puede tomar cero o uno argumentos : en todos casos, el objeto instanciado es el primer argumento y si hay un argumento es el segundo de un operador binario; sino es un operador unario.

Un operador definido afuera de clases, al nivel global, toma uno o dos argumentos (según si es unario o binario).

Operadores : sintaxis

Para llamar al operador :

```
int main() {
  Image a(100,100);
  Image b(200,200);
  Image c = a + b;
  cout << c.width << " " << c.height << endl;
};
```

Aquí, se llama el método “operador +” desde el objeto a.

Operadores unarios

Sin estar exhaustivo :

`++, --, &, !, +, -, ~`

Operadores binarios

Muchos operadores binarios pueden estar sobrecargados :

`+ , - , * , / , % , & , ^ , | , << , >> , += , -= , *= , /= , %= , ^= , &= , |= , >>= , <<= == , != , < , > , <= , >= , && , ||`

y cada uno está libre de darle el "sentido" que desea... Hasta los operadores de comparación pueden tomar sentido completamente distinto (y regresar otra cosa que booleano). Los argumentos también pueden ser heterogéneos.

Operadores

Dos maneras de sobrecargarlos : hacerles métodos de la clase o funciones globales (eventualmente declaradas friend).

```
class Image {
public:
  int width;
  int height;
  Image(int w,int h) {
    width=w;height=h;
  };
  Image operator+(const Image &otra);
  friend Image &operator+=(Image &img1,
                           const Image &img2);
};
```

Operadores

Función global :

```
Image &operator+=(Image &img1,const Image &img2) {
  int w = (img1.width>img2.width)?img1.width:img2.wid
  img1.width  = w;
  img1.height+= img2.height;
  return img1;
};
```

Uso igual...

Argumentos : ¿qué tipos ?

Usar los mismos criterios que con funciones :

- Usar referencias const cada vez que se sabe que el argumento no estará modificado en el operador.
- Referencias normales sólo si el objeto está modificado (+=,...).
- Eventualmente verificar (en el cuerpo del operador) que las referencias pasadas como argumentos no son iguales : puede generar problemas...

Argumentos : ¿qué tipos ?

El riesgo al pasar referencias iguales es:

```
a = a;
```

¿Peligro?

Valores de regreso ?

- El valor de regreso debe de ser coherente y intuitivo : $+$ regresará típicamente un nuevo objeto.
- Operadores lógicos : que regresen int o, mejor, bool.
- Los operadores de asignación deben de poder ser usados en cadena como $a = b = c$, y, sobre todo se suele efectuar operaciones sobre el resultado de $a = b$, entonces regresar una referencia no-const

Valores de regreso ?

```
Image &Image::operator=(const Image &img2) {
        ...
        return *this;
}
...
a = b = c;
(d=e).doSomething();
```

Operadores de incremento

Problema, se necesita distinguir las declaraciones/definiciones de pre-operación de post-operación, entonces se adoptó lo siguiente :
Pre-operación : “++a”

```
operator++(a)
```

Post-operación : “a++”

```
operator++(a, int)
```

(operador unario/binario)

Operadores de incremento

- Pre-operación : se hace la operación sobre el objeto y se regresa $^*\text{this}$.
- Post-operación : se necesita regresar una copia del objeto antes de llevar a cabo la operación.

Post-operaciones ligeramente mas costosas. . .

Regreso de valor

- Puede ser preferible regresar por valor con const : el resultado de un operador binario no tiene que servir para invocar métodos (ya que de todos modos vamos a perder referencia a este objeto).
- Regresar inteligentemente :

```
Image img(w,h); // Constructor
return img; // Copy const. hacia valor de regreso
            // + destructor
```

Preferir :

```
return Image(w,h); // Directamente el de regreso
```

3 operaciones en un caso, 1 sola en el otra

Operador de uso especial

- Notar que unos operadores no pueden estar definidos como funciones globales : asignaciones, operador corchete.
- El operador corchete toma un valor entero de entrada y regresa una referencia non-const, para poder escribir,

```
img[i] = ...;
```

Operador coma

Como en el caso de la incrementación, dos formas para diferenciar:

- coma después como miembro :

```
const Image::Image &operator,(const Image &i) const {
...
return(*this);
}
```

- coma antes como función global :

```
const Image &operator,(int,const Image &i) {
...
}
```

Operador coma

Uso muy exótico...

```
Image a,b,c;
a,b;
1,c;
```

Limitaciones

- Operador de acceso a miembro . no sobrecargable.
- Operador de dereferencia de un apuntador a miembro .* no sobrecargable.
- No se puede crear operadores nuevos.

Operadores no miembros : ¿cuándo ?

- En general, mejor hacerlo método de la clase.
- Si no se puede trabajar sobre la clase del primer objeto, entonces mejor usar una función global.

Ejemplo :

```
ostream&
operator<<(ostream& os, const Image& img) {
  os << "(" << img.width << "␣" <<  img.height << ")"
}
```

Operadores no miembros : ¿cuándo ?

Operadores miembros, en particular, para :

- Operadores unarios.
- Operadores relacionados con memoria, apuntadores (=,[],->,...).
- Operador combinado a una asignación (+=,-=...).

Lo demás puede ser definido fuera de la clase...

Operador de asignación

Cuidado al no mezclar las dos acepciones del "=" :

```
Image a = b;
c = b;
```

- En el primer caso, no es una afectación normal, es una creación de objetos (constructor por copia). En el segundo caso, sí es una asignación (operador=).
- En practica, el constructor por copia y el operador = van a compartir código (un método copy por ejemplo).

Operador de asignación

Notar que el mismo "=" puede servir a invocar otros constructores que el constructor por copia :

```
Image a = 200;
```

Puede tener sentido si está definido un constructor $\mathrm{Image(int)}$.

Operador de asignación y copia

En el caso de la presencia de un apuntador a otro objeto :

```
class ImageProc {
  workStruct *imgtmp;
}
```

¿Cómo definir la copia y la asignación?

La opción uno es re-copiar todo en un nuevo apuntador, pero cuidado a la auto-asignación

```
a = a;
```

Pero ¿qué tal que sí queremos compartir los datos?

Operador de asignación y copia

Existe otra opción que es el conteo de las referencias : la idea es usar el mismo apuntador para todas las instancias copiadas de una instancia $\mathrm{ImageProc}$, y de memorizar dentro de la clase $\mathrm{workStruct}$ el numero de objetos refiriéndose a ella.

Operador de asignación y copia

```
class workStruct {
  int data;
  int refCounter;
  int id;
  static int ids;
  workStruct() : refCounter(1),id(ids) {ids++;};
  workStruct(const workStruct &ws) {};
public:
  void attach() {++refCounter;};
  void detach() {
    if (--refCounter==0) {
      delete this;
      cout << "Object " << id
           << " destroyed" << endl;
    }
  };
  static workStruct *allocate() {return new workStruct;};
  void print() {cout << "Image " << id << ", refs : "
                << refCounter << endl;}
};
```

Operador de asignación y copia

- Sólo se puede crear un workStruct con allocate().
- Las clases externas comparten los workStruct usando attach() y rdetach().
- refCounter cuenta cuantas clases externas usan esa estructura.
- Automáticamente, se libera la memoria cuando ya no hay objetos usando el objeto alocado dinámicamente.

Operador de asignación y copia

```
class ImageProc {
  workStruct *ws;
public:
  ImageProc() {
    ws = workStruct::allocate();
  };
  ImageProc(const ImageProc &otro) {
    ws = otro.ws;
    ws->attach();
  };
  ImageProc &operator=(const ImageProc &otro){
    ws->detach();
    ws= otro.ws;
    ws->attach();
  };
  ~ImageProc() {
    ws->detach();
  };
  void print() {ws->print();}
};
```

Operador de asignación y copia

```
ImageProc ip1; ip1.print();
ImageProc ip2(ip1); ip2.print();
ImageProc *ip3 = new ImageProc;
ip3->print();
*ip3 = ip1; ip3->print();
ImageProc *ip4 = new ImageProc(ip2);
ip4->print();
delete ip3; ip2.print();
delete ip4; ip2.print();
```

Operador de asignación y copia

```
Image 0, refs : 1
Image 0, refs : 2
Image 1, refs : 1
Object 1 destroyed
Image 0, refs : 3
Image 0, refs : 4
Image 0, refs : 3
Image 0, refs : 2
Object 0 destroyed
```

Operador de asignación y copia

De esa manera, un objeto copiado (por constructor de copia o por asignación) puede eventualmente sobrevivir después de que el objeto original haya sido destruido.

Operador de asignación y copia

Si ahora quiero hacer modificaciones en los datos, sin que las otras copias lo tengan que ver, tengo que pasar un modo “clone” de la estructura : compartir los datos entre varias instancias es útil sobre todo para aplicaciones de lectura !

Operador de asignación y copia

Por default las clases vienen con un operador de asignación que hace lo mismo que el constructor por copia : las mismas remarcas se aplican (en particular que puede ser útil redefinirlos).

Operadores : aplicación a smart pointers

Ya que tenemos operadores $->$, $++$ podemos encapsular verdaderos apuntadores dentro de una clase que haga más que simples apuntadores :

```
class imageContainer {
  vector<Image*> contain;
public:
  void add(Image* img) { contain.push_back(img); }
  friend class SmartPointer;
};
```

Operadores : aplicación a smart pointers

```
class SmartPointer {
  imageContainer& ic;
  int index;
public:
  SmartPointer(imageContainer& imgc) : ic(imgc) {
    index = 0;
  }
  // Incrementa (prefijo)
  bool operator++() {
    if(index >= ic.contain.size()) return false;
    if(ic.contain[++index] == 0) return false;
    return true;
  }
  bool operator++(int) {
    return operator++();
  }
```

Operadores : aplicación a smart pointers

```
Image* operator->() const {
  if (ic.contain[index] != 0)
    return ic.contain[index];
  return NULL;
 }
};
```

Operadores : aplicación a smart pointers

```
int main() {
  const int sz = 10;
  Image img[sz];
  imageContainer ic;
  for(int i = 0; i < sz; i++)
    ic.add(&img[i]); // Fill it up
  SmartPointer sp(ic); // Create an iterator
  do {
    sp->doSomething();
  } while(sp++);
}
```

Operadores : conversión automática

Para poder compilar cosas como :

```
class myClass1 {
        ...
};
class myClass2 {
        ...
};
void fonc(myClass1 &o1);
int main() {
        myClass2 o2;
        fonc(o2);
        ...
```

Conversión implícita.

Operadores : conversión automática

Manera 1 : un constructor adecuado

```
class myClass1 {
      myClass1(const myClass2 &o2);
};
```

Se puede impedir de que se use la forma implícita y forzar el usuario a usar el constructor explícitamente :

```
class myClass1 {
      explicit myClass1(const myClass2 &o2);
};
int main() {
      myClass2 o2;
      fonc(myClass1(o2));
      ...
```

Pero eso sólo si tienes acceso a la clase myClass1...

Operadores : conversión automática

Manera 2 : por operador en la clase que queremos convertir automáticamente

```
class myClass2 {
        operator myClass1() const {
                myClass1 o1;
                ...
                return o1;
        };
};
```

Permite también conversiones a tipos elementales.

Operadores : globales vs. locales

Una ventaja de llamar operadores globales que es consecuencia de lo precedente : la conversión automática de los tipos puede estar hecha de los dos lados.

```
class Integer {
      Integer (int i=0) {...};
      Integer &operator+(const Integer &otro) {...}
};
Integer &operator-(const Integer &o1,
                   const Integer &o2) { ... };
Integer i1,i2,i3;
i2 = i1 + 1;
i3 = 1  + i1; // NO !
i2 = i1 - 1;
i3 = 1  - i1; // OK !
```

P.D. : se busca los operadores mas simples primero ! (1-1)

Operadores de conversión

No combinar varias formas de conversión ! Puede llevar ambigüedades : no sabrá cual método usar.

```
class myClass1 {
      myClass1(const myClass2 &o2);
};
class myClass2 {
      operator myClass1() const;
}
...
myClass2 o2;
fonc(o2);
```

¿Cuál usa? Mismo tipo de problemas si se define varios operadores de conversión y si se sobrecarga paralelamente las funciones.

Operadores new/delete

Los operadores de alocación/liberación de memoria relativos a la clase definida pueden estar sobrecargados

```
#include <new>
class Image {
        void *operator new(size_t sz) {
            ...
            return ::new char[sz];
        }
        void *operator new[](size_t sz) {
            ...
            return ::new char[sz];
        }
```

El primero es llamado a cada construcción de un nuevo objeto por new Image (en el montículo), el segundo al crear arreglos con new Image[n].

Operadores new/delete

Similarmente, se definen los operadores de deletion :

```
#include <new>
class Image {
        void operator delete(void* p) {
            ...
            return ::delete p;
        }
        void operator delete[](void *p) {
            ...
            return ::delete []p;
        }
```

Operadores new/delete

Notar que se puede añadir argumentos adicionales, si uno quiere, al operador new :

```
#include <new>
class Image {
public:
        void *operator new(size_t sz,int x) {
            ...
            return ::new char[sz];
        }
};
Image *img = new(4) Image(100,100);
```

Operadores new/delete

En todos casos se va a necesitar los operadores new/delete globales que pueden estar sobrecargados :

```
void* operator new(size_t sz) {
   printf("operator new: %d Bytes\n", sz);
   void* m = malloc(sz);
   if(!m) puts("out of memory");
   return m;
}
void operator delete(void* m) {
   puts("operator delete");
   free(m);
}
```