

Metodología de la Programación

Tema 5. Clases II: Sobrecarga de operadores

Departamento de Ciencias de la Computación e I.A.



Curso 2011-12

Contenido del tema

- 1 Introducción
- 2 Mecanismos de sobrecarga de operadores
 - Sobrecarga como función externa: Ejemplo operator+
 - Sobrecarga como función miembro: Ejemplo operator+
- 3 El operador de asignación
- 4 La clase mínima
- 5 Operadores << y >>
 - Sobrecarga del operador <<
 - Sobrecarga del operador >>
 - Sobrecarga del operador << con una función amiga
- 6 Operador de indexación

Introducción

Introducción I

- C++ permite usar un conjunto de operadores con los tipos predefinidos que hace que el código sea muy **legible y fácil de entender**.
- Por ejemplo, la expresión:

$$a + \frac{b \cdot c}{d \cdot (e + f)}$$

se calcularía en C++ con `a+(b*c)/(c*(e+f))`

- Si usamos un tipo que no dispone de esos operadores escribiríamos:
`Suma(a, Divide(Producto(b,c), Producto(c, Suma(e+f))))`
que es más engorroso de escribir y entender.

Introducción

Introducción II

- C++ permite **sobrecargar** casi todos sus operadores en nuestras propias clases, para que podamos usarlos con los objetos de tales clases.
- Para ello, definiremos un método o una función cuyo nombre estará compuesto de la palabra `operator` junto con el operador correspondiente. Ejemplo: `operator+()`.
- Esto permitirá usar la siguiente sintaxis para hacer cálculos con objetos de nuestras propias clases:

```
Polinomio p, q, r;  
// ...  
r= p+q;
```
- No puede modificarse la sintaxis de los operadores (número de operandos, precedencia y asociatividad).
- No deberíamos tampoco modificar la semántica de los operadores.

Operadores que pueden sobrecargarse

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

- Los operadores que no pueden sobrecargarse son:

.	.*	::	?:	sizeof
---	----	----	----	--------

- Al sobrecargar un operador no se sobrecargan automáticamente operadores relacionados.

Por ejemplo, al sobrecargar + no se sobrecarga automáticamente +=, ni al sobrecargar == lo hace automáticamente !=.

Sobrecarga como función externa

Sobrecarga como función externa

Consiste en añadir una función externa a la clase, que recibirá dos objetos (o uno para operadores unarios) de la clase y devolverá el resultado de la operación.

```
Polinomio operator+(const Polinomio &p1, const Polinomio &p2);
```

- Cuando el compilador encuentre una expresión tal como `p+q` la interpretará como una llamada a la función `operator+(p,q)`
- Incluso podríamos sobrecargar el operador aunque los dos operandos sean de tipos distintos:
 - Suma de Polinomio con float: `pol+3.5`

```
Polinomio operator+(const Polinomio &p1, float f);
```
 - Suma de float con Polinomio: `3.5+pol`

```
Polinomio operator+(float f, const Polinomio &p1);
```

Sobrecarga como función externa

```
Polinomio operator+(const Polinomio &p1, const Polinomio &p2){
    int gmax=(p1.getGrado()>p2.getGrado())?
        p1.getGrado():p2.getGrado();
    Polinomio resultado(gmax);
    for(int i=0;i<=gmax;++i){
        resultado.setCoeficiente(i,
            p1.getCoeficiente(i)+p2.getCoeficiente(i));
    }
    return resultado;
}

int main(){
    Polinomio p1, p2, p3;
    ... // dar valores a coeficientes de p2 y p3
    p1 = p2 + p3; // equivalente a p1 = operator+(p2,p3);
}
```

Sobrecarga como función miembro

Sobrecarga como función miembro

Consiste en añadir un método a la clase, que recibirá un objeto (o ninguno para operadores unarios) de la clase y devolverá el resultado de la operación.

```
Polinomio Polinomio::operator+(const Polinomio &p) const;
```

- Cuando el compilador encuentre una expresión tal como `p+q` la interpretará como una llamada al método `p.operator+(q)`
- También podríamos sobrecargar así el operador con un operando de tipo distinto:
 - Suma de Polinomio con float: `pol+3.5`

```
Polinomio Polinomio::operator+(float f) const;
```
 - Sin embargo no es posible definir así el operador para usarlo con expresiones del tipo: `3.5+pol`

Sobrecarga como función miembro

```
Polinomio Polinomio::operator+(const Polinomio &pol) const{
    int gmax=(this->getGrado()>pol.getGrado())?
        this->getGrado():pol.getGrado();
    Polinomio resultado(gmax);
    for(int i=0;i<=gmax;++i){
        resultado.setCoeficiente(i,
            this->getCoeficiente(i)+pol.getCoeficiente(i));
    }
    return resultado;
}

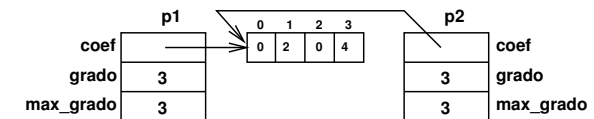
int main(){
    Polinomio p1, p2, p3;
    ... // dar valores a coeficientes de p2 y p3
    p1 = p2 + p3; // equivalente a p1 = p2.operator+(p3);
}
```

El operador de asignación

- En el siguiente código, la sentencia de asignación no funciona bien, ya que hace que p1 y p2 compartan la misma memoria dinámica al no haberse definido el método operator=.
- Cuando se ejecuta el destructor de p2 se produce un error al intentar liberar la memoria dinámica que liberó el destructor de p1.

```
class Polinomio {
private:
    float *coef;
    int grado;
    int max_grado;
public:
    Polinomio(int maxGrado=10);
    ~Polinomio();
    ...
};

int main(){
    Polinomio p1, p2;
    p1.setCoeficiente(3,4);
    p1.setCoeficiente(1,2);
    p2=p1;
    cout<<"Polinomio p1:"<<endl;
    p1.print();
    cout<<"Polinomio p2:"<<endl;
    p2.print();
}
```



El operador de asignación: primera aproximación

```
void operator=(const Polinomio &pol);
```

- Cuando realizamos una asignación del tipo p=q, el compilador lo interpreta como la llamada p.operator(q).
- Para evitar una copia innecesaria de q, pasamos el parámetro por referencia añadiendo const.
- En una asignación p=q se da valor a un objeto que ya estaba construido (*this ya está construido).
- En el constructor de copia se da valor a un objeto que está por construir.
- Por ello, en el operador de asignación debemos empezar liberando la memoria dinámica alojada en *this.
- El resto es idéntico al constructor de copia.

El operador de asignación: primera aproximación

```
void Polinomio::operator=(const Polinomio &pol){
    delete[] this->coef;
    this->max_grado=pol.max_grado;
    this->grado=pol.grado;
    this->coef=new float[this->max_grado+1];
    for(int i=0; i<=max_grado; ++i)
        this->coef[i]=pol.coef[i];
}
```

- Podemos ver que coincide con el constructor de copia, excepto en la primera línea.

El operador de asignación: primera aproximación

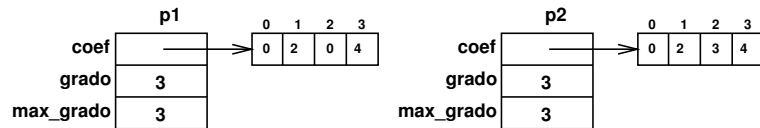
```

class Polinomio {
private:
    float *coef;
    int grado;
    int max_grado;
public:
    Polinomio(int maxGrado=10);
    ~Polinomio();
    ...
    void operator=(const Polinomio &pol);
};

void Polinomio::operator=(const Polinomio &pol){
    delete[] this->coef;
    this->max_grado=pol.max_grado;
    this->grado=pol.grado;
    this->coef=new float[this->max_grado+1];
    for(int i=0; i<=max_grado; ++i)
        this->coef[i]=pol.coef[i];
}

int main(){
    Polinomio p1, p2;
    p1.setCoeficiente(3,4);
    p1.setCoeficiente(1,2);
    p2=p1;
    cout<<"Polinomio p1:"<<endl;
    p1.print();
    cout<<"Polinomio p2:"<<endl;
    p2.print();
    p2.setCoeficiente(2,3);
    cout<<"Polinomio p1:"<<endl;
    p1.print();
    cout<<"Polinomio p2:"<<endl;
    p2.print();
}

```



El operador de asignación: segunda aproximación

```
Polinomio& operator=(const Polinomio &pol);
```

- Recordemos que el operador de asignación puede usarse de la siguiente forma: $p=q=r=s$.
- C++ evalúa la expresión anterior de derecha a izquierda, de forma que lo primero que realiza es $r=s$.
- El resultado de esta última expresión ($r=s$) es el objeto que queda a la izquierda (r), que se usa para evaluar el siguiente operador de asignación (asignación a q).
- Por tanto $operator=$ debe devolver el mismo tipo de la clase (Polinomio en este caso).
- Para que la llamada a $r.operator(s)$ devuelva el objeto r es necesario que la devolución sea por referencia.

El operador de asignación: segunda aproximación

```

Polinomio& Polinomio::operator=(const Polinomio &pol){
    delete[] this->coef;
    this->max_grado=pol.max_grado;
    this->grado=pol.grado;
    this->coef=new float[this->max_grado+1];
    for(int i=0; i<=max_grado; ++i)
        this->coef[i]=pol.coef[i];
    return *this;
}

```

- Como podemos ver, el método devuelve (por referencia) el objeto actual.

El operador de asignación: implementación final

```
Polinomio& operator=(const Polinomio &pol);
```

- En el caso de realizar una asignación del tipo $p=p$ nuestro operador de asignación no funcionaría bien.
- En tal caso, dentro del método $operator=$, $*this$ y pol son el mismo objeto.

```

Polinomio& Polinomio::operator=(const Polinomio &pol){
    if(&pol!=this){
        delete[] this->coef;
        this->max_grado=pol.max_grado;
        this->grado=pol.grado;
        this->coef=new float[this->max_grado+1];
        for(int i=0; i<=max_grado; ++i)
            this->coef[i]=pol.coef[i];
    }
    return *this;
}

```

El operador de asignación: esquema genérico

```
CLASE& operator=(const CLASE &p);
```

- En una clase que tenga datos miembro que usen memoria dinámica, éste sería el esquema genérico que debería tener operator=.

```
CLASE& CLASE::operator=(const CLASE &p)
{
    if (&p!=this) { // Si no es el mismo objeto
        // Si *this tiene memoria dinamica -> liberarla
        // Copiar p en *this (reservar nueva memoria y copiar)
    }
    return *this; // Devolver referencia a *this
}
```

La clase mínima

- En una clase, normalmente construiremos un constructor por defecto.
- Cuando la clase tiene datos miembro que usen memoria dinámica, añadiremos el destructor, constructor de copia y operador de asignación.

```
class Polinomio {
private:
    float *coef; // Array con los coeficientes
    int grado; // Grado de este polinomio
    int max_grado; // Maximo grado permitido en este polinomio
public:
    Polinomio(); // Constructor por defecto
    Polinomio(const Polinomio &p); // Constructor de copia
    ~Polinomio(); // Destructor
    Polinomio& operator=(const Polinomio &pol);
    void setCoeficiente(int i, float c);
    float getCoeficiente(int i) const;
    int getGrado() const;
};
```

Funciones miembro predefinidas

C++ proporciona una implementación por defecto para el constructor por defecto, destructor, constructor de copia y operador de asignación.

- Si no incluimos el **constructor por defecto**, C++ proporciona uno con cuerpo vacío.
- Si no incluimos el destructor, C++ proporciona uno con cuerpo vacío.
- Si no incluimos el constructor de copia, C++ proporciona uno que hace una copia de cada dato miembro llamando al constructor de copia de la clase a la que pertenece cada uno.
- Si no incluimos el operador de asignación, C++ proporciona uno que hace una asignación de cada dato miembro de la clase.

Sobrecarga del operador <<

- Podemos sobrecargar el operador << para mostrar un objeto usando la sintaxis cout << p (equivalente a cout.operator<<(p)).
- Puesto que no podemos añadir un método a la clase ostream (a la que pertenece cout), sobrecargaremos este operador con una función externa.

```
ostream& operator<<(ostream &flujo, const Polinomio &p){
    flujo<<p.getCoeficiente(p.getGrado()); // Termino de grado mayor
    if(p.getGrado()>0)
        flujo<<"x^"<<p.getGrado();
    for(int i=p.getGrado()-1;i>=0;--i){ // Recorrer resto de terminos
        if(p.getCoeficiente(i)!=0.0){ // Si el coeficiente no es 0.0
            flujo<<" + "<<p.getCoeficiente(i); // lo imprimimos
            if(i>0) cout<<"x^"<<i;
        }
    }
    flujo<<endl;
    return flujo;
}
```

Sobrecarga del operador <<

```
ostream& operator<<(ostream &flujo, const Polinomio &p){
    flujo<<p.getCoficiente(p.getGrado()); // Termina de grado mayor
    if(p.getGrado()>0)
        flujo<<"x^"<<p.getGrado();
    for(int i=p.getGrado()-1;i>=0;--i){ // Recorrer resto de terminos
        if(p.getCoficiente(i)!=0.0){ // Si el coeficiente no es 0.0
            flujo<<" " <<p.getCoficiente(i); // lo imprimimos
            if(i>0) cout<<"x^"<<i;
        }
    }
    flujo<<endl;
    return flujo;
}
```


- La función hace una devolución por referencia del flujo (ostream&).
- Esto se hace para poder usar sentencias como las siguientes:

```
Polinomio p1, p2;
... // Dar valor a coeficientes de p1 y p2
cout << p1;
cout << p1 << p2;
cout << p1 << p2 se evalua de izquierda a derecha:
(cout << p1) << p2
```

Sobrecarga del operador <<: Ejemplo

```
ostream& operator<<(ostream &flujo, const Polinomio &p){
    flujo<<p.getCoficiente(p.getGrado()); // Imprimimos termino de grado mayor
    if(p.getGrado()>0)
        flujo<<"x^"<<p.getGrado();
    for(int i=p.getGrado()-1;i>=0;--i){ // Recorremos el resto de terminos
        if(p.getCoficiente(i)!=0.0){ // Si el coeficiente no es 0.0
            flujo<<" " <<p.getCoficiente(i); // lo imprimimos
            if(i>0) cout<<"x^"<<i;
        }
    }
    flujo<<endl;
    return flujo;
}

int main(){
    Polinomio p1,p2;
    p1.setCoficiente(3,4);
    p1.setCoficiente(1,2);
    p2=p1;
    p2.setCoficiente(5,3);
    cout<<p1<<p2<<endl;
}
```



Sobrecarga del operador >>

- También podemos sobrecargar el operador >> para leer un objeto usando la sintaxis cin >> p (equivalente a cin.operator>>(p)).
- De nuevo, puesto que no podemos añadir un método a la clase istream (a la que pertenece cin), sobrecargaremos este operador con una función externa.

```
istream& operator>>(std::istream &flujo, Polinomio &p){
    int g;
    float v;
    do{
        flujo>> v >> g; //Introducir en la forma "valor grado"
        if(g>=0){
            p.setCoficiente(g,v);
        }
    }while(g>=0);
    return flujo;
}
```

Sobrecarga del operador >>

```
istream& operator>>(std::istream &flujo, Polinomio &p){
    int g;
    float v;
    do{
        flujo>> v >> g; //Introducir en la forma "valor grado"
        if(g>=0){
            p.setCoficiente(g,v);
        }
    }while(g>=0);
    return flujo;
}
```

- De nuevo vemos que la función hace una devolución por referencia del flujo (istream&).
- Esto se hace para poder usar sentencias como las siguientes:


```
Polinomio p1, p2;
cin >> p1;
cin >> p1 >> p2;
```

- cin >> p1 >> p2 se evalua de izquierda a derecha: (cin >> p1) >> p2

Sobrecarga del operador >>: Ejemplo

```
istream& operator>>(std::istream &flujo, Polinomio &p){
    int g;
    float v;
    do{
        flujo>> v >> g; // Los coeficientes se introducen en la forma "coeficiente grado"
        if(g>=0){ // Se introduce grado<0 para terminar
            p.setCoeficiente(g,v);
        }
    }while(g>=0);
    return flujo;
}

int main(){
    Polinomio p1;
    cout<<"Introduce polinomio \"coeficiente grado\" con 0 -1 para terminar: ";
    cin>>p1;
    cout<<"Polinomio="<<p1;
}
```



Sobrecarga del operador << con una función amiga

```
class Polinomio {
    float *coef; // Array con los coeficientes
    int grado; // Grado de este polinomio
    int max_grado; // Maximo grado permitido en este polinomio
public:
    ...
    friend ostream& operator<<(ostream &flujo, const Polinomio &p)
private:
    void inicializa();
};

ostream& operator<<(ostream &flujo, const Polinomio &p){
    flujo<<p.coef[p.grado]; // Termina de grado mayor
    if(p.grado>0)
        flujo<<"x^"<<p.grado;
    for(int i=p.grado-1;i>=0;--i){ // Recorrer resto de terminos
        if(p.coef[i]!=0.0){ // Si el coeficiente no es 0.0
            flujo<<" + "<<p.coef[i]<<"x^"<<i;
        }
    }
    flujo<<endl;
    return flujo;
}
```

Operador de indexación I

- La función operator[] () permite sobrecargar el operador de indexación.
- Debe realizarse usando un método de la clase con un parámetro (índice) que podría ser de cualquier tipo.
- De esta forma podremos cambiar la sintaxis:

```
x = p.getCoeficiente(i);
```

por esta otra:

```
x = p[i];
```

Operador de indexación II

- Una primera aproximación podría ser:

```
float Polinomio::operator[](int i){
    assert(i>=0); assert(i<=grado);
    return coef[i];
}
```

- Pero, si queremos cambiar la sintaxis:

```
p.setCoeficiente(i, x);
```

por esta otra:

```
p[i] = x;
```

necesitamos modificarlo:

```
float& Polinomio::operator[](int i){
    assert(i>=0); assert(i<=grado);
    return coef[i];
}
```

Operador de indexación III

- Por último, para poder usar este operador con un Polinomio constante, como por ejemplo en el siguiente código:

```
void funcion(const Polinomio p){  
    ...  
    x = p[i];  
    ...  
}
```

debemos definir también la siguiente versión del método:

```
const float& Polinomio::operator[] (int i) const{  
    assert(i>=0); assert(i<=grado);  
    return coef[i];  
}
```