Metodología de la Programación

Tema 5. Clases II: Sobrecarga de operadores

Andrés Cano Utrera
(acu@decsai.ugr.es)
Departamento de Ciencias de la Computación e I.A.

Curso 2019-2020

Contenido del tema

Contenido del tema

Introducción a la sobrecarga de operadores

- 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:

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

se calcularía en C++ con:

```
resultado = a+(b*c)/(c*(e+f))
```

- Si usamos un tipo que no dispone de esos operadores escribiríamos:

```
resultado = Suma(a,Divide(Producto(b,c),Producto(c,Suma(e,f))))
```

que es más engorroso de escribir y entender.

Introducción a la sobrecarga de operadores

- 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	new[]	delete		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 !=.

Contenido del tema

Contenido del tema

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);
}
```


Contenido del tema

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);
}
```


Contenido del tema

Sobrecarga como función miembro o externa

- La sobrecarga de un operador con una **función miembro** puede hacerse si tenemos acceso al código fuente de la clase y el primer operando es del tipo de la clase.
  Ejemplo: Para sumar dos polinomios, podemos sobrecargar `operator+` en la clase `Polinomio` con una función miembro, pues tenemos acceso a su implementación.

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

- El lenguaje obliga a que los operadores `()`, `[]`, `->` y el operador de asignación, sean implementados como **funciones miembro**.

Sobrecarga como función miembro o externa

- Si el primer operando debe ser un objeto de una clase diferente, debemos sobrecargarlo como **función externa**.
  Ejemplo: El operador + para concatenar un string con un `Polinomio` lo implementaremos con una función externa.

```
string operator+(const string& cadena, const Polinomio& pol){
...
}

int main(){
    Polinomio p;
    string s1="Polinomio: ", s2;
    ...
    s2 = s1 + p; // equivale a s2 = operator+(s1, p);
}
```

- También, si el primer operando debe ser un dato de un tipo primitivo, debemos sobrecargarlo como **función externa**.

```
Polinomio operator+(int i, const Polinomio& pol){
...
}

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


Sobrecarga como función miembro o externa

Directrices para elegir entre miembro y no-miembro

Según el libro de Rob Murray, C++ Strategies & Tactics , Addison Wesley, 1993, página 47.

Operador	Uso recomendado
Todos los operadores unarios	miembro
= () [] -> ->*	**debe** ser miembro
+= -= /= *= ^= &= |= %= >>= <<=	miembro
El resto de operadores binarios	no miembro

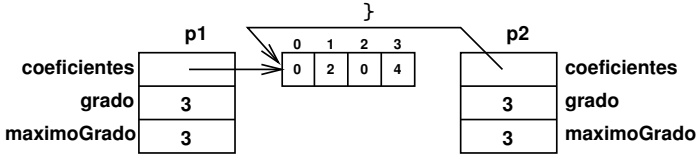
Contenido del tema

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 maximoGrado;
    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.imprimir(); cout << endl;
    cout<<"Polinomio p2:"<<endl;
    p2.imprimir();
}
```



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 memoría 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->coeficientes;
    this->maximoGrado=pol.maximoGrado;
    this->grado=pol.grado;
    this->coeficientes=new float[this->maximoGrado+1];
    for(int i=0; i<=maximoGrado; ++i)
        this->coeficientes[i]=pol.coeficientes[i];
}
```

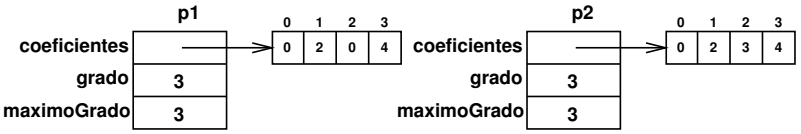
- Podemos ver que coincide con el constructor de copia, excepto en la primera línea.

El operador de asignación: primera aproximación

```
class Polinomio {
    private:
        float *coeficientes;
        int grado;
        int maximoGrado;
    public:
        Polinomio(int maxGrado=10);
        ~Polinomio();
        ...
        void operator=(const Polinomio &pol);

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

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



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++ evalua 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->coeficientes;
    this->maximoGrado=pol.maximoGrado;
    this->grado=pol.grado;
    this->coeficientes=new float[this->maximoGrado+1];
    for(int i=0; i<=maximoGrado; ++i)
        this->coeficientes[i]=pol.coeficientes[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->coeficientes;
        this->maximoGrado=pol.maximoGrado;
        this->grado=pol.grado;
        this->coeficientes=new float[this->maximoGrado+1];
        for(int i=0; i<=maximoGrado; ++i)
            this->coeficientes[i]=pol.coeficientes[i];
    }
    return *this;
}
```


El operador de asignación: esquema genérico

Esquema genérico del operador de asignación

En una clase que tenga datos miembro que usen memoria dinámica, el esquema genérico del operador de asignación (operator=) sería el siguiente:

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


Contenido del tema

La clase mínima

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

```
class Polinomio {
    private:
        float *coeficientes;     // Array con los coeficientes
        int grado;       // Grado de este polinomio
        int maximoGrado;  // Máximo 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 ningún constructor, C++ proporciona el **constructor por defecto** que tiene un 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.

Contenido del tema

Contenido del tema

Sobrecarga del operador <<

Operador << (operador de salida)

Se usa para enviar el contenido de un objeto a un flujo de salida (por ej. cout)

- 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`), usamos una **función externa**.


```
ostream& operator<<(ostream& flujo, const Polinomio& p){
    flujo << p.getCoeficiente(p.getGrado());//Mostrar término grado mayor
    if(p.getGrado()>1)
        flujo << "x^" << p.getGrado();
    else if (p.getGrado()==1)
        flujo << "x";
    for(int i=p.getGrado()-1;i>=0;--i){//Recorrer resto de términos
        if(p.getCoeficiente(i)!=0.0){     //Si el coeficiente no es 0.0
            if(p.getCoeficiente(i)>0.0) //imprimir coeficiente positivo
                flujo << " + " << p.getCoeficiente(i);
            else                        //imprimir coeficiente negativo
                flujo << " - " << -p.getCoeficiente(i);
            if(i>1)
              flujo << "x^" << i;
            else if (i==1)
              flujo << "x";
        }
    }
    return flujo;
}
```


Sobrecarga del operador <<

- 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 de uso

```
ostream& operator<<(ostream& flujo, const Polinomio& p){
    flujo << p.getCoeficiente(p.getGrado());//Mostrar término grado mayor
    if(p.getGrado()>1)
        flujo << "x^" << p.getGrado();
    else if (p.getGrado()==1)
        flujo << "x";
    for(int i=p.getGrado()-1;i>=0;--i){//Recorrer resto de términos
        if(p.getCoeficiente(i)!=0.0){     //Si el coeficiente no es 0.0
            if(p.getCoeficiente(i)>0.0) //imprimir coeficiente positivo
                flujo << " + " << p.getCoeficiente(i);
            else                        //imprimir coeficiente negativo
                flujo << " - " << -p.getCoeficiente(i);
            if(i>1)
              flujo << "x^" << i;
            else if (i==1)
              flujo << "x";
        }
    }
    return flujo;
}
int main(){
    Polinomio p1,p2;
    p1.setCoeficiente(3,4);
    p1.setCoeficiente(1,2);
    p2=p1;
    p2.setCoeficiente(5,3);
    cout<<p1<<p2<<endl;
}
```

Contenido del tema

Sobrecarga del operador >>

Operador >> (operador de entrada)

Se usa para leer el contenido de un objeto desde un flujo de entrada (por ej. cin).

- 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**.

Sobrecarga del operador >>

```
istream& operator>>(std::istream &flujo, Polinomio &p){
    int g;
    float v;

    p.clear();
    do{
        flujo>> v >> g;//Introducir coeficientes en la forma "coeficiente grado"
        if(g>=0){      // Se introduce grado<0 para terminar
            p.setCoeficiente(g,v);
        }
    }while(g>=0);
    return flujo;
}

void Polinomio::clear(){
    if(coef)
        delete[] coef;
    grado=0;
    max_grado=10;
    coef=new float[max_grado+1];
    for(int i=0; i<=max_grado; ++i)
        coef[i]=0.0;
}
```


Sobrecarga del operador >>

- De nuevo, el método devuelve por referencia el 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 de uso

```
istream& operator>>(std::istream &flujo, Polinomio &p){
    int g;
    float v;

    p.clear();
    do{
        flujo>> v >> g;//Introducir coeficientes 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;
}
```


Contenido del tema

Sobrecarga del operador << con una función amiga

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


Sobrecarga del operador << con una función amiga

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

Contenido del tema

Operador de indexación

Operador de indexación

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

- Una primera aproximación podría ser:

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


Operador de indexación

- 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 constante del método:

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

- Podemos implementar la versión constante del método de manera que no sea necesaria la copia del resultado al punto de llamada:

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

Operador de indexación

- La versión final de la implementación de este operador quedaría como:

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

int main(){
  Polinomio p1;
  float x;
  ...
  const Polinomio p2=p1;
  x=p2[j]; // Usa const float& Polinomio::operator[](int i) const
  p1[i]=x; // Usa float& Polinomio::operator[](int i)
}
```


Contenido del tema

Operadores de asignación compuestos

Operadores de asignación compuestos

Son los operadores +=, -=, *=, /=, %=, ^=, &=, |=, »=, «=

- Tener implementado el operador + y el operador = no supone la existencia automática del operador +=, y así con el resto: debemos implementarlo de **forma explícita**.
- Estos operadores deben devolver una referencia al objeto usado en la llamada, para así poder hacer por ejemplo:

```
p3 = (p1 += p2);
```

- Implementación como función miembro:

```
Polinomio& Polinomio::operator+=(const Polinomio& pol){
    *this = *this + pol;
    return *this;
}
```

- Es posible también implementarlos como función externa:

```
Polinomio& operator+=(Polinomio& pol1, const Polinomio& pol2){
    pol1 = pol1 + pol2;
    return pol1;
}
```


Contenido del tema

Operadores relacionales

Operadores relacionales

Son los operadores binarios ==, !=, <, >, <= y >=, que devuelven un valor booleano.

- Se usan cuando es necesario establecer una ***relación de orden*** entre los objetos de la clase.
- El definir una parte de los operadores no implica que los demás lo estén de forma automática.
  Ejemplo: si definimos el operador ==, el operador != no estará definido de forma automática.

Operadores relacionales

Ejemplo: operador < en Polinomio

pol1 < pol2 si pol1 tiene grado menor que pol2, o, si son del mismo grado, su coeficiente máximo es menor que el de pol2.

```
bool Polinomio::operator<(const Polinomio& pol) const{
   bool menor = this->getGrado()<pol.getGrado() ? true : false;
   if (!menor){
      bool iguales = this->getGrado()==pol.getGrado() ? true : false;
      if(iguales){
         menor = coeficientes[this->getGrado()]<
                 pol.coeficientes[this->getGrado()] ? true:false;
      }
   }
   return menor;
}
```


Contenido del tema

Operadores de incremento y decremento (++ y --)

Operadores de incremento y decremento

Son operadores unarios que tienen dos versiones: ***pre*** y ***pos***, tanto para incremento como para decremento.

```
Polinomio& Polinomio::operator++(){
   *this = *this + 1;
   return *this;
}
Polinomio& Polinomio::operator--(){
   *this = *this - 1;
   return *this;
}
int main(){
   Polinomio pol;
   ...
   ++pol;
   ...
   --pol;
}
```

Operadores de incremento y decremento (++ y --)

Operadores de posincremento y posdecremento

Los nombres de las funciones para los operadores ***pos*** coinciden con los ***pre***.
Por ello, el estándar de C++ propone que:

- Cuando el compilador encuentra `++obj`, se genera una llamada a `obj.operator++()`.
- Cuando el compilador encuentra `obj++`, se genera una llamada a `obj.operator++(0)`. En este caso se añade un valor entero a la llamada, que no se usa para nada, pero que sirve para distinguirla de la anterior.

Operadores de incremento y decremento (++ y --)

```
Polinomio Polinomio::operator++(int valor){
   Polinomio aux(*this);
   *this = *this + 1;
   return aux;
}
Polinomio Polinomio::operator--(int valor){
   Polinomio aux(*this);
   *this = *this - 1;
   return aux;
}
int main(){
   Polinomio pol;
   ...
   pol++;
   ...
   pol--;
}
```

¡Cuidado!

La devolución en este caso debe hacerse por valor. ¿Por qué?

Contenido del tema

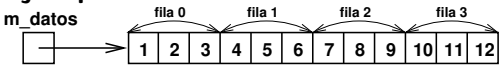
Operador de llamada a función

Operador de llamada a función

Es la función `operator()` que obligatoriamente se implementará como función miembro.
Puede implementarse con cualquier número de parámetros (podemos tener varias versiones de este operador).

Operador de llamada a función

Ejemplo:




```
class Matriz {
      double* m_datos;
      int m_filas, m_columnas;
   public:
      Matriz(int nf, int nc){
         m_filas=nf;
         m_columnas=nc;
         m_datos = new double[m_filas*m_columnas];
      }
      double& operator() (int fila, int columna){
         assert(fila>=0 && fila<m_filas && columnas >=0 && columna<m_columnas);
         return m_datos[fila*m_columnas + columna];
      }
      const double& operator() (int fila, int columna) const{
         assert(fila>=0 && fila<m_filas && columnas >=0 && columna<m_columnas);
         return m_datos[fila*m_columnas + columna];
      }
}
```


Operador de llamada a función

```
int main(){
   Matriz m(4,3);
   ...
   cout<<m(3,2)<<endl;
   m(3,2) = 7.4;
}
```