Metodología de la Programación

Tema 5. Clases II: Sobrecarga de operadores

Departamento de Ciencias de la Computación e I.A.

ETSIIT Universidad de Granada

Curso 2017-18

Contenido del tema

Contenido del tema

Introducción

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

Introducción

- 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))`

Introducción

- 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

- C++ permite **sobrecargar** casi todos sus operadores en nuestras clases, para que podamos usarlos con sus objetos.

Introducción

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

Introducción

- C++ permite **sobrecargar** casi todos sus operadores en nuestras clases, para que podamos usarlos con sus objetos.
- 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;
// ¡Ojo! Esta expresión involucra al operador + y al operador =
r = p+q;
```

Introducción

- C++ permite **sobrecargar** casi todos sus operadores en nuestras clases, para que podamos usarlos con sus objetos.
- 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;
// ¡Ojo! Esta expresión involucra al operador + y al operador =
r = p+q;
```

- No puede modificarse la sintaxis de los operadores (número de operandos, precedencia y asociatividad).

Introducción

- C++ permite **sobrecargar** casi todos sus operadores en nuestras clases, para que podamos usarlos con sus objetos.
- 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;
// ¡Ojo! Esta expresión involucra al operador + y al operador =
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

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)`

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:

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);
```

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.obtenerGrado()>p2.obtenerGrado())?
             p1.obtenerGrado() : p2.obtenerGrado();
  Polinomio resultado(gmax);
  for(int i=gmax; i>=0; --i)
    resultado.asignarCoeficiente(i,
        p1.obtenerCoeficiente(i)+p2.obtenerCoeficiente(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 método de la clase

Sobrecarga como método (función miembro) de la clase

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 podemos 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 método de la clase

```
Polinomio Polinomio::operator+(const Polinomio &pol) const{
  int gmax = (this->obtenerGrado()>pol.obtenerGrado()) ?
             this->obtenerGrado():pol.obtenerGrado();
  Polinomio resultado(gmax);
  for(int i=0;i<=gmax;++i)
    resultado.asignarCoeficiente(i,
        this->obtenerCoeficiente(i)+pol.obtenerCoeficiente(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);
}
```

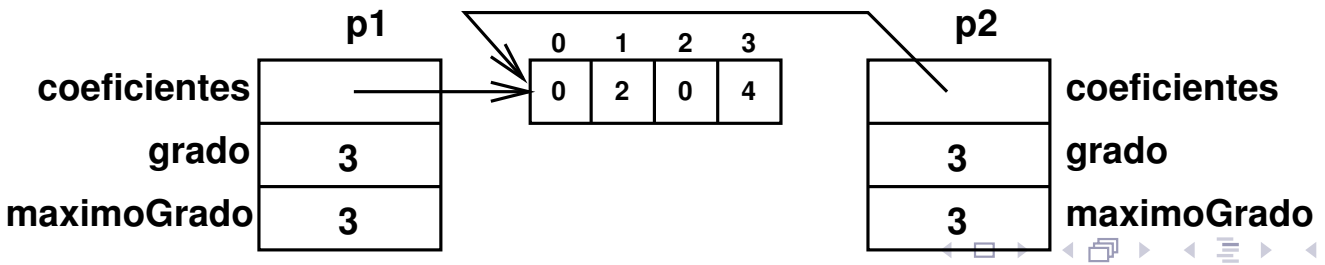
Contenido del tema

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.asignarCoeficiente(3,4);
    p1.asignarCoeficiente(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)`.

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`.

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`.
- Diferencia entre el constructor y el operador de asignación:

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`.
- Diferencia entre el constructor y el operador de asignación:
  - En una asignación `p=q` se da valor a un objeto que ya estaba construido (`this` apunta a un objeto existente).

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`.
- Diferencia entre el constructor y el operador de asignación:
  - En una asignación `p=q` se da valor a un objeto que ya estaba construido (`this` apunta a un objeto existente).
  - En el constructor de copia se da valor a un objeto que está por construir.

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`.
- Diferencia entre el constructor y el operador de asignación:
  - En una asignación `p=q` se da valor a un objeto que ya estaba construido (`this` apunta a un objeto existente).
  - En el constructor de copia se da valor a un objeto que está por construir.
- Por ello, en el operador de asignación debemos empezar liberarando la memoría dinámica alojada en `*this`.

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`.
- Diferencia entre el constructor y el operador de asignación:
  - En una asignación `p=q` se da valor a un objeto que ya estaba construido (`this` apunta a un objeto existente).
  - En el constructor de copia se da valor a un objeto que está por construir.
- Por ello, en el operador de asignación debemos empezar liberarando la memoría dinámica alojada en `*this`.
- El resto del código 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];
}
```

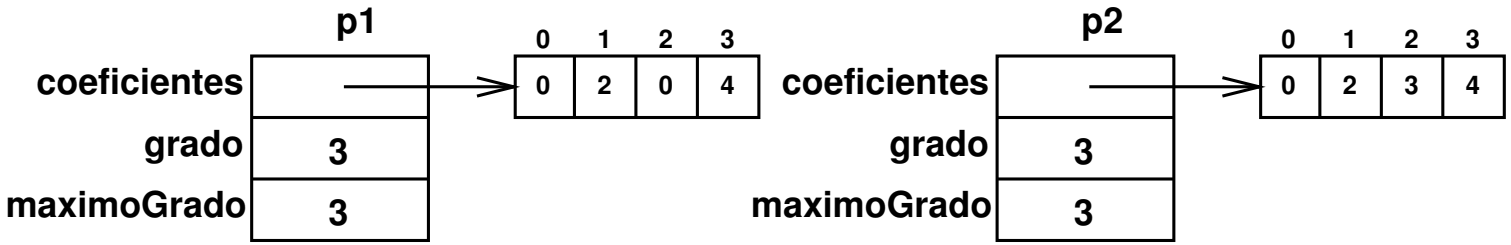
- Podemos ver que coincide con el constructor de copia, excepto en la primera línea.

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.asignarCoeficiente(3,4);
    p1.asignarCoeficiente(1,2);
    p2=p1;
    cout<<"Polinomio p1:"<<endl;
    p1.print();
    cout<<"Polinomio p2:"<<endl;
    p2.print();
    p2.asignarCoeficiente(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;`.

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 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`).

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).

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`, y no una copia del objeto, 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 I

```
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: implementación final II

Teniendo en cuenta que el operador de asignación y el constructor de copia comparten código, podemos reescribirlos usando un método auxiliar:

```
//Método auxiliar privado copia los datos del polinomio
//pasado como argumento en *this
void Polinomio::copiar(const Polinomio &pol){
  maximoGrado = pol.maximoGrado;
  grado = pol.grado;
  coeficientes = new float[maximoGrado+1];
  for(int i=0; i<=maximoGrado; ++i)
    coeficientes[i] = pol.coeficientes[i];
//  memcpy(coeficientes, p.coeficientes, (maximoGrado+1)*sizeof(float));
}
```

El operador de asignación: implementación final III

```
//Constructor de copia
Polinomio::Polinomio(const Polinomio &pol){
  copiar(pol);
}

//Operador de asignación
Polinomio& Polinomio::operator=(const Polinomio &pol){
  if(&pol!=this){
    delete[] this->coeficientes;
    copiar(pol);
  }
  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
}
```

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;  // 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 asignarCoeficiente(int i, float c);
        float obtenerCoeficiente(int i) const;
        int obtenerGrado() 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.

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.

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.

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

Sobrecarga de operadores compuestos

Habiendo estudiado la sobrecarga de un operador binario como + y del operador de asignación, podemos plantearnos también sobrecargar **operadores compuestos** como +=.

```
//Operador += como miembro de la clase
Polinomio & Polinomio::operator+=(const Polinomio &pol){
  //Aprovechamos los operadores + y = para evitar repetir código
  *this = *this + pol;
  return *this;
}
```

- Obsérvese que lo hemos implementado como un método de la clase. Podríamos haberlo hecho también como una función externa a la clase.
- Si queremos permitir usar este operador con operandos mixtos (Polinomio + float o float + Polinomio), habrá casos (float + Polinomio) en los que habrá que implementarlo como función externa (no tenemos acceso a la clase float).

Contenido del tema

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 &pol);
```

- La función hace una devolución por referencia del flujo (`ostream&`). Esto se hace para poder encadenar el operador:

```
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;
```

```
ostream& operator<<(ostream &flujo, const Polinomio &p){
  //Imprimimos el primer coeficiente
  flujo << p.obtenerCoeficiente(p.obtenerGrado());
  if(p.obtenerGrado()>1)
    flujo << "x^" << p.obtenerGrado();
  else if (p.obtenerGrado()==1)
    flujo << "x";
  //No hay else. Si grado == 0 no imprimimos x
  //Imprimimos el resto de coeficientes
  for(int i=p.obtenerGrado()-1; i>=0; i--){
    if (p.obtenerCoeficiente(i) != 0){
      if (p.obtenerCoeficiente(i) > 0)
        flujo << " + " << p.obtenerCoeficiente(i);
      else if(p.obtenerCoeficiente(i)<0)
        flujo << " - " << -p.obtenerCoeficiente(i);
      if (i>1)
        flujo << "x^" << i;
      else if (i==1)
        flujo << "x";
      //No hay else. En caso de que i==0 no mostramos nada
    }
    //No hay else. Si el coeficiente es 0 no mostramos nada
  }
  flujo << endl;
  return flujo;
}
```

Sobrecarga del operador <<: Ejemplo

```
ostream& operator<<(ostream &flujo, const Polinomio &p){

   ...

}

int main(){
  Polinomio p1,p2;
  p1.asignarCoeficiente(3,4);
  p1.asignarCoeficiente(1,2);
  p2=p1;
  p2.asignarCoeficiente(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)`).
- Como no podemos añadir un método a la clase `istream` (a la que pertenece `cin`), sobrecargamos el operador con una función externa.

```
istream& operator»(istream &flujo, Polinomio &pol);
```

- De nuevo, el método devuelve por referencia el flujo (`istream&`). Esto se hace para poder encadenar el operador:

```
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;//Introducir coeficientes en la forma "coeficiente grado"
    if(g>=0)       // Se introduce grado<0 para terminar
      p.asignarCoeficiente(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 I

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

ostream& operator<<(ostream &flujo, const Polinomio &p){
  //Imprimimos el primer coeficiente
  flujo << p.coeficientes[p.grado];
  if(p.grado>1)
    flujo << "x^" << p.grado;
  else if (p.grado==1)
    flujo << "x";
```

Sobrecarga del operador << con una función amiga II

```
//Imprimimos el resto de coeficientes
  for(int i=p.grado-1; i>=0; i--){
    if (p.coeficientes[i] != 0){
      if (p.coeficientes[i] > 0)
        flujo << " + " << p.coeficientes[i];
      else if(p.coeficientes[i]<0)
        flujo << " - " << -p.coeficientes[i];
      if (i>1)
        flujo << "x^" << i;
      else if (i==1)
        flujo << "x";
    }
  }
  flujo << endl;
  return flujo;
}
```

Contenido del tema

Operador de indexación

- El método `operator[]()` permite sobrecargar el operador de indexación.

Operador de indexación

- El método `operator[]()` permite sobrecargar el operador de indexación.
- Debe realizarse usando un método de la clase con un parámetro, que proporciona el índice.

Operador de indexación

- El método `operator[]()` permite sobrecargar el operador de indexación.
- Debe realizarse usando un método de la clase con un parámetro, que proporciona el índice.
- De esta forma podremos cambiar la sintaxis:

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

por esta otra:

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

Operador de indexación

- El método `operator[]()` permite sobrecargar el operador de indexación.
- Debe realizarse usando un método de la clase con un parámetro, que proporciona el índice.
- De esta forma podremos cambiar la sintaxis:

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

por esta otra:

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

- Una primera aproximación podría ser:

```
float Polinomio::operator[](int i){
  ...
  return coeficientes[i];
}
```

Operador de indexación

- Pero, si queremos cambiar la sintaxis:

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

  por esta otra:

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

  necesitamos modificarlo para que devuelva una referencia al coeficiente indexado:

  ```
  float& Polinomio::operator[](int i){
    ...
    return coeficientes[i];
  }
  ```

Operador de indexación

- Pero, si queremos cambiar la sintaxis:

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

  por esta otra:

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

  necesitamos modificarlo para que devuelva una referencia al coeficiente indexado:

  ```
  float& Polinomio::operator[](int i){
    ...
    return coeficientes[i];
  }
  ```

¡Importante!

Éste es el primer método de la clase que devuelve una referencia, no al objeto, sino a uno de sus datos miembro.

Operador de indexación

- 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{
  ...
  return coeficientes[i];
}
```

Operador de indexación: primera aproximación

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

Operador de indexación: primera aproximación

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

Observaciones:

- No acepta la posibilidad de indexar un elemento mayor que `maximoGrado`.

Operador de indexación: primera aproximación

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

Observaciones:

- No acepta la posibilidad de indexar un elemento mayor que `maximoGrado`.
- Podemos revisar el código para ampliar el vector de coeficientes si se indexa una posición mayor que `maximoGrado`.

Operador de indexación: primera aproximación

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

Observaciones:

- No acepta la posibilidad de indexar un elemento mayor que `maximoGrado`.
- Podemos revisar el código para ampliar el vector de coeficientes si se indexa una posición mayor que `maximoGrado`.
- De esa forma podremos devolver una referencia a un objeto válido.

Operador de indexación: segunda aproximación

```
float& Polinomio::operator[](int ind){
  assert(ind>=0);
  if (ind>maximoGrado){
    //Se ha indexado fuera del vector. Lo ampliamos
    //Reservamos el nuevo espacio
    float *aux = new float[ind];
    //Copiamos los coeficientes existentes
    memcpy(aux, coeficientes, maximoGrado+1);
    //Liberamos el antiguo vector de coeficientes
    delete[] coeficientes;
    //Actualizamos el puntero
    coeficientes = aux;
    //Ponemos a 0 el resto de coeficientes
    for(int i=maximoGrado+1; i<=ind; i++)
      coeficientes[i] = 0.0;
  }
  //No hay else. Si ind<=maximoGrado sólo tenemos
  //que devolver el coeficiente
  return coeficientes[ind];
}
```

Operador de indexación: segunda aproximación

Observaciones:

- Puesto que se devuelve una componente del vector de coeficientes, ésta podría ser modificada una vez terminada la ejecución del método, sin actualizar debidamente el dato miembro `grado`.

Operador de indexación: segunda aproximación

Observaciones:

- Puesto que se devuelve una componente del vector de coeficientes, ésta podría ser modificada una vez terminada la ejecución del método, sin actualizar debidamente el dato miembro `grado`.
- Además, en la versión `const` del operador no podemos operar de esta manera, ya que no podemos modificar el objeto.

Operador de indexación: segunda aproximación

Observaciones:

- Puesto que se devuelve una componente del vector de coeficientes, ésta podría ser modificada una vez terminada la ejecución del método, sin actualizar debidamente el dato miembro `grado`.
- Además, en la versión `const` del operador no podemos operar de esta manera, ya que no podemos modificar el objeto.
- El problema en la versión `const` del método no es difícil de resolver:

```
const float& Polinomio::operator[](int ind) const{
  assert(ind>=0);
  const static float cero = 0.0;
  if (ind>maximoGrado)
    //Se ha indexado fuera del vector de coeficientes
    //Devolvemos una referencia a 0.0 constante
    return cero;
  else
    //Si ind<=maximoGrado sólo tenemos que devolver el coeficiente
  return coeficientes[ind];
}
```

Operador de indexación: segunda aproximación

Observaciones:

- Pero seguimos teniendo el gravísimo problema de que el grado del polinomio puede verse alterado **desde fuera del código del operador**, por lo que la representación interna del polinomio será inconsistente.

Operador de indexación: segunda aproximación

Observaciones:

- Pero seguimos teniendo el gravísimo problema de que el grado del polinomio puede verse alterado **desde fuera del código del operador**, por lo que la representación interna del polinomio será inconsistente.
- La única alternativa que podemos plantearnos con esta representación interna (vector dinámico) es eliminar el dato miembro `grado` y revisar el método `obtenerGrado()` para que calcule el grado del polinomio en lugar de consultar el dato miembro.

Operador de indexación: segunda aproximación

Observaciones:

- Pero seguimos teniendo el gravísimo problema de que el grado del polinomio puede verse alterado **desde fuera del código del operador**, por lo que la representación interna del polinomio será inconsistente.
- La única alternativa que podemos plantearnos con esta representación interna (vector dinámico) es eliminar el dato miembro `grado` y revisar el método `obtenerGrado()` para que calcule el grado del polinomio en lugar de consultar el dato miembro.
- Esto supone un evidente coste en eficiencia a cambio de poder proporcionar al usuario de la clase un potente operador como es `[]`, que mejora sustancialmente su uso.

Clase polinomio: implementación revisada I

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

#ifndef Polinomio_h
#define Polinomio_h

class Polinomio{
private:
  //Array con los coeficientes del polinomio
  float * coeficientes;
  //Máximo grado posible: limitación debida a la implementación
  //de la clase: el array de coeficientes tiene un tamaño limitado
  int maximoGrado;
public:
  //Constructor sobrecargado: por defecto y con parámetro
  Polinomio(int maximoGrado=10);
  //Constructor de copia
  Polinomio(const Polinomio& p);
  //Destructor
  ~Polinomio();
```

Clase polinomio: implementación revisada II

```
  int grado() const;
  Polinomio & operator=(const Polinomio & pol);
  Polinomio operator+(const Polinomio & pol) const;
  Polinomio operator+(float f) const;
  float& operator[](int i);
  const float& operator[](int i) const;
  Polinomio & operator+=(const Polinomio & pol);
private:
  void copiar(const Polinomio &pol);
};

Polinomio operator+(float f, const Polinomio &pol);

ostream& operator<<(ostream &flujo, const Polinomio &p);
istream& operator>>(istream &flujo, Polinomio &p);

#endif
```

Clase polinomio: implementación revisada III

```
#include <iostream>
#include <assert.h>
#include <string.h>
#include "Polinomio.h"

using namespace std;

Polinomio::Polinomio(int maximoGrado) {
  // Si maximoGrado no es positivo terminamos el programa
  assert(maximoGrado > 0);
  // Si el valor de maximoGrado es correcto,
  // se asigna su valor al dato miembro
  this->maximoGrado = maximoGrado;
  // Se inicializan los demás datos miembro
  // Se reserva espacio para el array de coeficientes
  coeficientes = new float[maximoGrado + 1];
  // Se inicializan a valor 0
  for (int i = 0; i <= maximoGrado; i++)
    coeficientes[i] = 0.0;
}
```

Clase polinomio: implementación revisada IV

```
//Constructor de copia
Polinomio::Polinomio(const Polinomio &pol){
  copiar(pol);
}

//Método auxiliar privado copia los datos del polinomio
//pasado como argumento en *this
void Polinomio::copiar(const Polinomio &pol){
  maximoGrado = pol.maximoGrado;
  coeficientes = new float[maximoGrado+1];
//  for(int i=0; i<=maximoGrado; ++i)
//    coeficientes[i] = pol.coeficientes[i];
  memcpy(coeficientes, pol.coeficientes, (maximoGrado+1)*sizeof(float));
}

//Destructor
Polinomio::~Polinomio(){
  delete []coeficientes;
}
```

Clase polinomio: implementación revisada V

```
int Polinomio::grado() const {
  int grado = maximoGrado;
  while(coeficientes[grado]==0.0 && grado>0)
    grado--;
  return grado;
}

//Operador de asignación
//Usa el método auxiliar copiar() para evitar repetir código
Polinomio& Polinomio::operator=(const Polinomio &pol){
  if(&pol!=this){ //Si no son el mismo objeto
    delete[] coeficientes; //Liberamos el vector de coeficientes
    copiar(pol);           //Copiamos
  }
  return *this;
}

//Sobrecarga del operador + como función externa
//para permitir la operación float + Polinomio
Polinomio operator+(float f, const Polinomio &pol){
  Polinomio resultado(pol);
```

Clase polinomio: implementación revisada VI

```
  resultado[0] = resultado[0] + f;
  return resultado;
}

//Sobrecarga del operador + como método de la clase
Polinomio Polinomio::operator+(const Polinomio &pol) const{
  int gmax = (this->grado()>pol.grado()) ? this->grado() : pol.grado();
  Polinomio resultado(gmax);
  for(int i=0;i<=gmax;++i)
    resultado[i] = (*this)[i] + pol[i];
  return resultado;
}

//Sobrecarga del operador +
//para permitir la operación Polinomio + float
Polinomio Polinomio::operator+(float f) const{
  Polinomio resultado(*this);
  resultado[0] = resultado[0] + f;
  return resultado;
}
```

Clase polinomio: implementación revisada VII

```
ostream& operator<<(ostream &flujo, const Polinomio &p){
  int grado = p.grado();
  //Imprimimos el primer coeficiente
  flujo << p[grado];
  if(grado>1)
    flujo << "x^" << grado;
  else if (grado==1)
    flujo << "x";
  //No hay else. Si grado == 0 no imprimimos x

  //Imprimimos el resto de coeficientes
  for(int i=grado-1; i>=0; i--){
    if (p[i] != 0){
      if (p[i] > 0)
        flujo << " + " << p[i];
      else if(p[i]<0)
        flujo << " - " << -p[i];
      if (i>1)
        flujo << "x^" << i;
      else if (i==1)
        flujo << "x";
```

Clase polinomio: implementación revisada VIII

```
      //No hay else. En caso de que i==0 no mostramos nada
    }
    //No hay else. Si el coeficiente es 0 no mostramos nada
  }
  flujo << endl;
  return flujo;
}

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

//Operador []
```

Clase polinomio: implementación revisada IX

```
float& Polinomio::operator[](int ind){
  assert(ind>=0);
  if (ind>maximoGrado){
    //Se ha indexado fuera del vector. Lo ampliamos
    //Reservamos el nuevo espacio
    float *aux = new float[ind+1];
    //Copiamos los coeficientes existentes
    memcpy(aux, coeficientes, (maximoGrado+1)*sizeof(float));
    //Liberamos el antiguo vector de coeficientes
    delete[] coeficientes;
    //Actualizamos el puntero
    coeficientes = aux;
    //Ponemos a 0 el resto de coeficientes
    for(int i=maximoGrado+1; i<=ind; i++)
      coeficientes[i] = 0.0;
    maximoGrado = ind;
  }
  //No hay else. Si ind<=maximoGrado sólo tenemos
  //que devolver el coeficiente
  return coeficientes[ind];
}
```

Clase polinomio: implementación revisada X

```
//Operador [], versión constante
const float& Polinomio::operator[](int ind) const{
  assert(ind>=0);
  const static float cero = 0.0;
  if (ind>maximoGrado)
    //Se ha indexado fuera del vector de coeficientes
    //Devolvemos una referencia a 0.0 constante
    return cero;
  else
    //Si ind<=maximoGrado sólo tenemos que devolver el coeficiente
    return coeficientes[ind];
}

//Operador +=
Polinomio & Polinomio::operator+=(const Polinomio &pol){
  //Aprovechamos los operadores + y = para evitar repetir código
  *this = *this + pol;
  return *this;
}
```