Metodología de la Programación

Tema 5. Sobrecarga de operadores

Departamento de Ciencias de la Computación e I.A.

Curso 2015-16

Contenido del tema I

Parte I: Conceptos básicos

Parte II: caso de estudio, clase Array

Parte I

Conceptos básicos

Índice

Objetivo

Sobrecarga de operadores:

tiene como objetivo permitir que los operadores de C++ puedan trabajar de forma directa con los objetos de nuestras propias clases. Así, las expresiones con objetos se parecen, en la medida de lo posible, a las expresiones formadas con datos de tipo primitivo.

Ejemplo

Imaginemos que a, b, c, d, e f y res son objetos de alguna clase (por ejemplo, la clase NumeroComplejo). Supongamos que deseamos programar una determinada operación en la que los objetos actúan como operandos:

$$res = a + \frac{b \times c}{d \times (e + f)}$$

Lo ideal sería poder escribir la siguiente sentencia y que el compilador asuma que las operaciones se están realizando con objetos.

$$res = a + (b * c)/(d * (e + f));$$

Solución hasta ahora:

Pero esto no puede hacerse directamente, ya que C++ no sabe cómo usar los operadores +, *, / y = (entre otros) sobre objetos (ya veremos que para algunos C++ ofrece comportamiento por defecto). De momento, la única alternativa viable consiste en usar la sintaxis propia de orientación a objetos:

```
res=a.suma((b.multiplicar(c)).divide(d.multiplicar(e.suma(f))));
```

Solución hasta ahora:

Pero esto no puede hacerse directamente, ya que C++ no sabe cómo usar los operadores +, *, / y = (entre otros) sobre objetos (ya veremos que para algunos C++ ofrece comportamiento por defecto). De momento, la única alternativa viable consiste en usar la sintaxis propia de orientación a objetos:

```
res=a.suma((b.multiplicar(c)).divide(d.multiplicar(e.suma(f))));
```

¿Es suficiente con esto?, ¿dónde podríamos tener problemas?

Solución hasta ahora: I

A todas luces el uso de la sintaxis propia de orientación a objetos produce una expresión más difícil de leer y entender:

$$res = a + (b * c)/(d * (e + f));$$

```
res=a.suma((b.multiplicar(c)).divide(d.multiplicar(e.suma(f))));
```

Solución con sobrecarga

C++ permite **sobrecargar** los operadores en nuestras propias clases, de forma que podamos usarlos con los objetos correspondientes.

Para ello, definiremos métodos (o funciones) cuyo nombre estará compuesto de la palabra `operator` junto con el símbolo del operador. Ejemplo: `operator+()`.

Solución con sobrecarga

Esto permitirá usar la siguiente sintaxis para manipular también objetos:

```
Polinomio p, q, r;
// ...
r = p+q;
```

Solución con sobrecarga

Aunque no seamos conscientes de ello, ya hemos usado la sobrecarga de operadores. Por ejemplo, el operador:

$<<$

se usa en dos contextos diferentes sin que haya confusión: operación de salida sobre flujos (con cout, por ejemplo) y para indicar la rotación a izquierda a nivel de bits. Igual ocurre con el operador $>>$.

Solución con sobrecarga I

La clase string ofrece también esta posibilidad.

```
#include <iostream>
#include <string>

using namespace std;

int main(){
  string cadena1("Hola, Pepe");
  string cadena2(", ");
  string cadena3("bien te veo... ");

  // Concatenacion mediante operador +
  cadena1+=cadena2+cadena3;
```

Solución con sobrecarga II

```
// Se muestra resultado
cout << cadena1 << endl;

// Tambien los operadores relacionales (menor)
if (cadena1 < cadena3){
  cout << "cadena1 menor que cadena3" << endl;
}

// Ahora igualdad
if (cadena1 == cadena3){
  cout << "cadena1 igual a cadena3" << endl;
}
```

Solución con sobrecarga III

```
  // Mayor
  if (cadena1 > cadena3){
    cout << "cadena1 mayor que cadena3" << endl;
  }

  // Distintos
  if (cadena1 != cadena3){
     cout << "cadena1 distinto de cadena3" << endl;
  }

  // Tambien podemos asignar cadenas de forma completa
  cadena2=cadena3;
  cout << "Ahora cadena2 debe ser igual a cadena3: " << endl;
  cout << "cadena2: " << cadena2 << endl;
  cout << "cadena3: " << cadena3 << endl;
}
```

Solución con sobrecarga IV

La salida producida es:

```
Hola, Pepe, bien te veo...                    // Concatenacion
cadena1 menor que cadena3                     // menor (longitud)
cadena1 distinto de cadena3                   // distintos
Ahora cadena2 debe ser igual a cadena3:       // Tras asignacion igualdad
cadena2: bien te veo...
cadena3: bien te veo...
```

Tratamiento de errores I

Aunque no forma parte del contenido del tema, consideraremos, a lo largo del mismo dos formas diferentes de tratamiento de posibles errores: aserciones y excepciones. Ambas permiten finalizar el programa en caso de error, pero de formas muy diferentes....

Índice

Operador con comportamiento por defecto

C++ ofrece comportamiento por defecto para el operador de asignación. Pero como ocurre en el caso del constructor de copia por defecto, no hace copia nueva del espacio de memoria asignado de forma dinámica. Si trabajamos con clases en que se use memoria dinámica y deseamos usar de forma correcta este operador tendremos que sobrecargarlo,como ahora veremos.

Nota: a veces los programas pueden no mostrar la salida esperada debido a optimizaciones que evitan algunas operaciones innecesarias. Se irán comentando a lo largo del tema, teniendo en cuenta que cada compilador aplica su propia política de optimizaciones y no todos se adhieren de forma completa al estándar de C++.

Operadores que pueden sobrecargarse

Debemos de tener claro qué operadores pueden sobrecargarse:

$+$	$-$	$*$	$/$	$\%$	$\hat{}$	$\&$	$\vert$
$\sim$	$<<$	$>>$	$=$	$+=$	$-=$	$*=$	$/=$
$\%=$	$\hat{}=$	$\&=$	$\vert=$	$>>=$	$<<=$	$==$	$!=$
$<$	$>$	$<=$	$>=$	$!$	$\&\&$	$\vert\vert$	$++$
$--$	$->*$	,	$->$	$[]$	$()$	new	delete
new[]	delete[]						

Y obviamente, también los que no pueden sobrecargarse:

.	.*	::	? :	sizeof

Operadores que pueden sobrecargarse

Al sobrecargar un operador no se sobrecargan automáticamente otros relacionados. Por ejemplo, al sobrecargar $+$ no se sobrecarga automáticamente $+=$, ni al sobrecargar $==$ lo hace $!=$.

Reglas y restricciones de la sobrecarga

Hay que tener en cuenta las siguientes consideraciones:

- no puede cambiarse la precedencia de los operadores
- no puede cambiarse la asociatividad de los operadores
- no puede cambiarse el carácter unario o binario de los operadores
- no pueden crearse nuevos operadores
- la sobrecarga debe ser natural, asignando una funcionalidad que esté realmente asociada con el significado del operador

Índice

Mecanismos de sobrecarga de operadores

Para analizar la forma en que puede realizarse la sobrecarga distinguiremos entre operadores binarios y unarios. Se analizan de forma abstracta estos dos tipos de operadores para hacer las consideraciones pertinentes antes de ver ejemplos de implementación.

Sobrecarga de operadores binarios

Consideremos una clase (por ejemplo, la clase Array) en la que queremos definir la operación de suma mediante la sobrecarga de operadores. De esta forma, dados dos objetos objeto1 y objeto2 de dicha clase queremos poder hacer

$$objeto1 \quad + \quad objeto2$$

Esta operación puede entenderse de dos formas diferentes:

- método de la clase: $objeto1.operator+(objeto2)$
- función ajena a la clase: $operator+(objeto1, objeto2)$

En ambos casos hemos prescindido de considerar qué devuelve (lo lógico es que fuese un objeto de la misma clase).

Sobrecarga como método

Pensemos en la clase Array. Al sobrecargar el operador como método de la clase, su declaración (en el archivo de cabecera) sería:

```
Array operator+(const Array &) const;
```

Con esto:

- indicamos que el objeto sobre el que se hace la llamada no se modifica
- el objeto pasado como argumento (sumando segundo) se pasa por referencia
- el resultado de la operación es un nuevo objeto de la clase

Sobrecarga como función externa

Consiste en añadir una función externa a la clase, que recibirá dos objetos de la clase y devolverá el resultado de la operación. La declaración del método sería:

```
Array operator+(const Array &, const Array &);
```

Consideraciones:

- al tener que acceder a los datos miembro de la clase Array, sin ser un método, debe acceder a través de los métodos de acceso disponibles en la clase
- esto puede ser costoso en tiempo, por la sobrecarga computacional necesaria para la gestión de las llamadas a los métodos de acceso
- para evitar esto suele hacerse que la función sea amiga de la clase (aquí sí tiene sentido usar este mecanismo aunque optaremos por hacer la sobrecarga como método de la clase)

Sobrecarga como función externa

Si deseamos poder operar con datos de tipo distinto (por ejemplo, para sumar un valor de tipo float a un Array; se incrementarían todos sus valores en el valor dado), la sobrecarga debería hacerse como función externa. Hemos de tener en cuenta que el valor float puede aparecer a la derecha y a la izquierda:

- suma de `Array` con `float`: `objeto+3.5`

```
Array operator+(const Array &, float valor);
```

- suma de `float` con `Array`: `3.5+objeto`

```
Array operator+(float valor, const Array &);
```

Si el primer operando es un valor, la opción de implementación mediante método miembro queda descartada.

Sobrecarga de operadores unarios

Consideremos que en la clase Array queremos definir la operación ! con sobrecarga de operadores. Usándolo sobre un objeto de la clase Array comprobaría si todos los valores son 0 (el significado de la operación debe definirlo el programador de la clase):

$$!objeto1$$

Esta operación puede entenderse de dos formas diferentes:

- método de la clase: $objeto1.operator!()$
- función ajena a la clase: $operator!(objeto1)$

La operación debería devolver un valor de tipo booleano.

Sobrecarga como método

Si se sobrecarga este operador en la la clase Array, la declaración del método (en el archivo de cabecera) sería:

```
bool operator!() const;
```

Con esto:

- indicamos que el objeto sobre el que se hace la llamada no se modifica
- el resultado de la operación es un valor booleano

Sobrecarga como función externa

Se añadiría una función externa a la clase, que recibirá un objeto como argumento y devuelve el resultado de la operación. La declaración del método sería:

```
bool operator!(const Array &);
```

Consideraciones:

- al tener que acceder a los datos miembro de la clase Array, sin ser un método, debe acceder a través de los métodos de acceso disponibles en la clase
- esto puede ser costoso en tiempo, por la sobrecarga computacional necesaria para la gestión de las llamadas a los métodos de acceso
- para evitar esto suele hacer que la función sea amiga de la clase

Parte II

Caso de estudio: clase Array

Índice

Clase Array

Para estudiar de forma práctica todos estos conceptos consideraremos una clase de referencia, como hicimos en el tema anterior. En este caso se trata de una clase Array propia, que evite algunos de los problemas de los arrays convencionales:

- controle el acceso a posiciones no válidas, evitando errores de ejecución
- permita asignar de forma completa los valores de un array
- permita usar los operadores relacionales sobre objetos de la clase, para determinar si son iguales, distintos, ...
- controlar de forma automática su tamaño
- .....

Índice

Estructura básica de la clase: archivo de cabecera I

Comenzaremos considerando la estructura básica de la clase, con los elementos ya estudiados hasta ahora: constructores y destructor básicamente. El archivo de cabecera sería el siguiente:

```
#ifndef ARRAY_H
#define ARRAY_H

#include <assert.h>
using namespace std;

class Array{
private:
   /**
    * Data miembro para almacenar el tam. del array
    */
   int size;
```

Estructura básica de la clase: archivo de cabecera II

```
    /**
     * Puntero para referenciar el espacio de memoria de almacenamiento
     */
    int *ptr;

public:
    /**
     * Se usa un parametro por defecto para usar un unico constructor
     * @param size tamaño deseado para el array, por defecto es 10
     */
    Array(int = 10);

    /**
     * Constructor de copia por defecto
     * @param otro objeto a copiar
     */
    Array(const Array &otro);
```

Estructura básica de la clase: archivo de cabecera III

```
    /**
     * Destructor
     */
    ~Array(){
       delete [] ptr;
    }

    /**
     * Metodo inline para devolver el tamaño del array
     * @return
     */
    inline int getSize() const{
       return size;
    }
};

#endif  /* ARRAY_H */
```

Estructura básica de la clase: archivo cpp I

```cpp
#include "Array.h"

/**
 * Constructor
 * @param size tamaño del array, 10 por defecto
 */
Array::Array(int size){
   // Se comprueba que el tamaño sea mayor que 0
   assert(size > 0);

   // Se asigna el valor de size
   this->size=size;

   // Se reserva espacio de memoria
   ptr=new int[size];

   // Se inicializa
   for(int i=0; i < size; i++){
      ptr[i]=0;
   }
}
```

Estructura básica de la clase: archivo cpp II

```
/**
 * Constructor de copia por defecto
 * @param otro objeto a copiar
 */
Array::Array(const Array &otro) {
   // Se asigna el tamaño
   size=otro.size;

   // Se reserva espacio
   ptr=new int[size];

   // Se inicializa con los valores de otro
   for(int i=0; i < size; i++){
      ptr[i]=otro.ptr[i];
   }
}
```

Estructura básica de la clase: probando aserciones I

```cpp
#include <cstdlib>
#include "Array.h"

using namespace std;

int main(int argc, char** argv) {
   // Se intenta crear un array de tamaño 0. Vemos que
   // ocurre con assert
   Array array0(0);
}
```

La salida obtenida es:

```
array: src/Array.cpp:9: Array::Array(int): Assertion ‘size > 0’ failed.
Abortado (‘core’ generado)
```

Estructura básica de la clase: evitando el core....

Está bien preparar el programa para que considere el caso en que el tamaño es cero o negativo, pero, ¿es necesario generar un core? ¿No sería mejor avisar al usuario de que ha habido algún problema y finalizar de forma ordenada, en caso de no poderse seguir ejecutando?

La solución recomendada pasa por el uso de **excepciones**.

Estructura básica de la clase: usando excepciones

Sólo incluiremos aquí los elementos que cambian con respecto de la versión anterior:

- el archivo de cabecera, para incluir el **include** de las excepciones
- el constructor donde se hace la comprobación del tamaño
- el método **main**

El include necesario es:

```
#include <stdexcept>
```

Las excepciones consideradas en este archivo se agrupan en errores lógicos (error lógico, error de dominio, argumento inválido, error de longitud, fuera de rango) y errores de ejecución (error de ejecución, error de rango, error de desbordamiento por exceso y por defecto).

Estructura básica de la clase: uso de excepciones I

El constructor sería ahora:

```
/**
 * Constructor
 * @param size tamaño del array, 10 por defecto
 */
Array::Array(int size){
   // Se comprueba que el tamaño sea mayor que 0
   if (size > 0){
      this->size=size;
   }
   else{
      throw invalid_argument("El tam. del array debe ser > 0");
   }

   // Se reserva espacio de memoria y se inicializa
   ptr=new int[size];
   for(int i=0; i < size; i++){
      ptr[i]=0;
   }
}
```

Estructura básica de la clase: uso de excepciones I

Y el método **main** quedaría:

```
#include <cstdlib>
#include <iostream>
#include "Array.h"

using namespace std;

int main(int argc, char** argv) {
   // Se intenta crear un array de tamaño 0. Vemos que
   // ocurre con assert
   Array *array;
   try{
      array=new Array(0);
   }catch(invalid_argument &exception){
      // Consideramos que el problema es muy grave y forzamos la salida
      // con exit..... pero podria continuarse si es posible
      cout << "Problema de ejecucion: " << exception.what() << endl;
      exit(-1);
   }
   cout << "No se sale del programa si todo OK o no hay exit...." << endl;
}
```

Estructura básica de la clase: uso de excepciones II

La salida producida ahora es:

```
Problema de ejecucion: El tam. del array debe ser > 0
```

Índice

Sobrecarga del operador <<

El objetivo de agregar esta funcionalidad es poder visualizar de forma cómoda el contenido completo de un array, con la siguiente sentencia:

```
cout << array << endl; // Array es un objeto....
```

Conviene tener en cuenta que:

- el operando que aparece en primer lugar es **cout**
- el segundo operando es el objeto a mostrar

Esto hace que la única implementación viable sea mediante una función externa (no puede ser método), que haremos amiga de la clase **Array**.

Sobrecarga del operador <<

La declaración del método se incluye dentro de la clase para indicar que se trata de una función amiga.

```
.....................................................................
class Array{
   /**
    * Sobrescritura del operador << como funcion externa amiga
    */
   friend ostream & operator<<(ostream &, const Array &);
private:
   /**
    * Data miembro para almacenar el tam. del array
    */
   int size;
.....................................................................
```

Sobrecarga del operador <<

Conviene hacer las siguientes observaciones:

- es una función amiga de la clase **Array**, para poder acceder a los datos miembros de la clase de forma sencilla

- el método devuelve una referencia a ostream para permitir que el resultado de la operación pueda usarse para otra nueva salida:

```
cout << array1 << array2;
```

Sobrecarga del operador << I

La implementación del método es (la haríamos en un archivo denominado **utils.cpp**, por ejemplo):

```
#include "Array.h"

/**
 * Sobrescritura del oprador <<
 * @param output
 * @param array
 * @return
 */
ostream & operator<<(ostream &output, const Array &array){
   output << endl;
   for(int i=0; i < array.size; i++){
      output << array.ptr[i] << " ";
   }
   output << endl;

   // Se devuelve output
   return output;
}
```

Sobrecarga del operador << I

El main necesario para probar las operaciones incluye las sentencias:

```
.....................................................
// Se crea un array de tamaño 7
Array enteros1(7);
// Se crea otro de tamaño 10, con el constructor por defecto
// usando el parametro por defecto
Array enteros2;

// Se visualiza el primer array
cout << "Enteros1 (7 elementos): " << enteros1 << endl;
cout << "Enteros2 (10 elementos): " << enteros2 << endl;

// Por la forma en que se implementa, tambien podemos hacerlo asi
cout << "Enteros1: " << enteros1 << "Enteros2: " << enteros2 << endl;
.....................................................
```

Aclaraciones

A tener en cuenta:

- el uso del constructor por defecto sólo puede hacerse si no se incluyen los paréntesis al declarar el objeto: Array array;

- la creación mediante **Array()** produce error de compilación, al menos en algunos compiladores. El de gnu lo acepta, pero el de mac no

Sobrecarga del operador <<

La salida obtenida es:

```
Enteros1 (7 elementos):
0 0 0 0 0 0 0

Enteros2 (10 elementos):
0 0 0 0 0 0 0 0 0 0

Enteros1:
0 0 0 0 0 0 0
Enteros2:
0 0 0 0 0 0 0 0 0 0
```

Índice

Sobrecarga del operador >>

El objetivo de agregar esta funcionalidad es poder asignar valores al array de forma cómoda, según se van escribiendo mediante el teclado:

```
cin >> array; // Array es un objeto....
```

Conviene tener en cuenta que:

- el operando que aparece en primer lugar es **cin**
- el segundo operando es el objeto al que se asignarán valores

Como ocurría con <<, la única implementación posible es mediante una función externa (no puede ser método), que haremos amiga de la clase por las razones ya consideradas previamente **Array**.

Sobrecarga del operador >>

La declaración de la función se incluye dentro de la clase para indicar que se trata de una función amiga.

```
.....................................................
class Array{
   /**
    * Sobrescritura del operador << como funcion externa amiga
    */
   friend ostream & operator<<(ostream &, const Array &);

   /**
    * Sobrescritura del operador >> como funcion externa amiga
    */
   friend istream & operator>>(istream &, Array &);

private:
   /**
    * Data miembro para almacenar el tam. del array
    */
   int size;
.....................................................
```

Sobrecarga del operador >>

Observaciones:

- es una función amiga de la clase **Array**, para poder acceder a los datos miembros de la clase de forma sencilla
- el argumento no se declara ahora como constante, ya que se modifica su contenido como resultado de la operación
- el método devuelve una referencia a istream para permitir que el resultado de la operación pueda usarse para otra nueva entrada:

```
cin >> array1 >> array2;
```

Sobrecarga del operador >> I

La implementación es (en utils.cpp, por ejemplo):

```
#include "Array.h"

.....................................................

/**
 * Sobrescritura del operador >>
 * @param input flujo de entrada
 * @param array array al que asignar los valores leidos
 * @return
 */
istream & operator>>(istream &input, Array &array){
   // Lectura de tantos valores como tamaño tiene el array
   for(int i=0; i < array.size; i++){
      input >> array.ptr[i];
   }

   // Se devuelve la referencia al flujo de entrada
   return input;
}
```

Sobrecarga del operador << I

El **main** necesario para probar las operaciones incluye las sentencias:

```
.....................................................
// Se crea un array de tamaño 7
Array enteros1(7);
// Se crea otro de tamaño 10, con el constructor por defecto
// usando el parametro por defecto
Array enteros2;

// Se prueba la lectura sobre el array de 7 elementos
cout << "\nLectura de array1 con tam.: " << enteros1.getSize() << endl;
cin >> enteros1;

// Se prueba la lectura sobre el array de 10 elementos
cout << "\nLectura de array2 con tam.: " << enteros2.getSize() << endl;
cin >> enteros2;

// El metodo de sobrescritura devuelve input & para poder hacer
// la lectura de la siguiente forma
cout << "\nIntroduzca " << enteros1.getSize() << " para enteros1 y " <<
        enteros2.getSize() << " para entero2" << endl;
```

Sobrecarga del operador << II

```
cin >> enteros1 >> enteros2;

// Se muestra el resultado mediante el operador <<
cout << "\nContenido de enteros1: ";
cout << enteros1;

cout << "\nContenido de enteros2: ";
cout << enteros2 << endl;
.....................................................
```

Sobrecarga del operador >>

La salida obtenida es:

```
.................................................
Lectura de array1 con tam.: 7
1 2 3 4 5 6 7

Lectura de array2 con tam.: 10
8 9 0 11 12 13 14 15 16 17

Introduzca 7 para enteros1 y 10 para entero2
1 2 3 4 5 6 7 8 9 10 11 12 13 14 15 16 17

Contenido de enteros1:
1 2 3 4 5 6 7

Contenido de enteros2:
8 9 10 11 12 13 14 15 16 17
.................................................
```

Índice

Sobrecarga del operador =

Importante: C++ ofrece una implementación por defecto para este operador. Como ocurría con el constructor de copia, la asignación por defecto no hace **copia dura** de los datos: si hay memoria dinámica ambos objetos la compartirán.

En segundo lugar hay que definir el efecto de la operación sobre la clase. El comportamiento deseado consiste en asignar el contenido completo de un array a otro:

```
// array1 y array2 son objetos ya existentes
// Al final de la asignacion ambos objetos tienen
// el mismo contenido y cada uno posee su propio
// espacio de memoria
array1 = array2;
```

Al ser ambos operandos objetos de la clase podemos optar por las dos alternativas ya vistas. Usaremos la implementación mediante método de la clase.

Sobrecarga del operador =

La declaración del método se hace en la parte pública del archivo de cabecera de la clase:

```
.....................................................................
   /**
    * Sobrescritura del operador =. Se hace que la referencia
    * devuelta sea constante para evitar que pueda hacerse
    * (obj1 = obj2) = obj3;
    * @param  otro sumando
    * @return resultado de la operacion
    */
   const Array &operator=(const Array &otro);
.....................................................................
```

Sobrecarga del operador $=$

Observaciones:

- el método devuelve una referencia constante para evitar que pueda aplicarse el operador de forma repetida: $(a = b) = c;$. Si deseamos que funcione de esta forma quitaríamos **const** del tipo de salida
- el método devuelve el objeto apuntado por **this**, ya que ha sido modificado por la operación
- el operador de asignación puede plantearse de la siguiente forma: en primer lugar se libera el espacio asociado a this y luego todas las operaciones coinciden con las del constructor de copia
- es importante evitar asignaciones del estilo $p = p$ ....

Sobrecarga del operador = I

La implementación es:

```
/**
 * Sobrescritura del operador =. Se hace que la referencia
 * devuelta sea constante para evitar que pueda hacerse
 * (obj1 = obj2) = obj3;
 * @param  otro sumando
 * @return resultado de la operacion
 */
const Array & Array::operator=(const Array &otro){
   // En primer lugar se evita la auto-asignacion
   // es decir, hacer objeto1 = objeto1. Solo se
   // opera si el objeto pasado como argumento es
   // distinto de otro
   if (&otro != this){
      // Se comprueba si es necesario redimensionar el array
      if (size != otro.size){
         // Se libera el espacio previo
         delete [] ptr;
```

Sobrecarga del operador = II

```
            // Se asigna el tamaño
            size=otro.size;

            // Se asigna el espacio de memoria
            ptr=new int[size];
        }

        // Se copian los valores de otro
        for(int i=0; i < size; i++){
              ptr[i]=otro.ptr[i];
        }
    }

    // Se devuelve este objeto
    return *this;
}
```

Sobrecarga del operador = I

Y probamos si funciona bien con un ejemplo (observad que se asigna a un array de 7 posiciones otro de 10):

```
.....................................................
// Recordemos que enteros1 tiene tam. 7 y enteros2 10
// Se hace la lectura sobre el array de 10 elementos
cout << "\nLectura de array2 con tam.: " << enteros2.getSize() << endl;
cin >> enteros2;

// Se prueba el operador recien creado
enteros1 = enteros2;

// Se muestra el resultado
cout << "enteros1: " << enteros1 << endl;
cout << "enteros2: " << enteros2 << endl;
.....................................................
```

Sobrecarga del operador =

La salida obtenida es:

```
.................................................
Lectura de array2 con tam.: 10
1 2 3 4 5 6 7 8 9 10
enteros1:
1 2 3 4 5 6 7 8 9 10

enteros2:
1 2 3 4 5 6 7 8 9 10
.................................................
```

Índice

Sobrecarga del operador +

El objetivo del operador + sobre arrays puede ser el de concatenar sus contenidos, actualizando el tamaño de forma adecuada:

```
array1 + array2; // array1 y array2 son objetos....
```

Al ser ambos operandos objetos de la clase tenemos dos alternativas de implementación: como método de la clase y como función externa. Optamos aquí por la primera: método de la clase.

Sobrecarga del operador +

La declaración del método se hace en la parte pública del archivo de cabecera de la clase:

```
.....................................................
   /**
    * Sobrescritura del operador +
    * @param otro sumando2
    * @return resultado de la suma
    */
   Array operator+(const Array &otro) const;
.....................................................
```

NOTA: podría devolverse una referencia, con lo que se evita una copia del objeto en la devolución del método. Optamos por esta versión porque las optimizaciones del compilador también la evitarán en la mayoría de los casos y no es estrictamente necesario para el funcionamiento del operador.

Sobrecarga del operador +

Observaciones:

- el método se declara como constante para indicar que el objeto sobre el que se hace la llamada no se modifica
- el método devuelve el objeto resultante de la suma

Sobrecarga del operador + I

```
/**
 * Sobrescritura del operador +
 * @param otro segundo sumando
 * @return resultado de la suma
 */
Array Array::operator+(const Array &otro) const{
   // Se crea el array de resultado
   Array *resultado=new Array(size+otro.size);

   // Se copian en primer lugar los valores de this y los de otro
   for(int i=0; i < size; i++){
      resultado->ptr[i]=ptr[i];
   }
   for(int i=0; i < otro.size; i++){
      resultado->ptr[size+i]=otro.ptr[i];
   }

   // Se devuelve el objeto creado
   return *resultado;
}
```

Sobrecarga del operador + I

El main necesario para probar las operaciones incluye las sentencias:

```
.....................................................
// Recordemos que enteros1 tiene tam. 7 y enteros2 10
// Se prueba la lectura sobre el array de 7 elementos
cout << "\nLectura de array1 con tam.: " << enteros1.getSize() << endl;
cin >> enteros1;

// Se prueba la lectura sobre el array de 10 elementos
cout << "\nLectura de array2 con tam.: " << enteros2.getSize() << endl;
cin >> enteros2;

// Se prueba la suma
Array resultadoSuma=enteros1 + enteros2;

// Se muestra el resultado
cout << "\nResultado de la suma de enteros1 y enteros2 " << endl;
cout << resultadoSuma << endl;
```

Sobrecarga del operador + II

```
// Incluso podriamos hacer una suma de tres elementos
Array resultadoFinal = resultadoSuma + enteros1 + enteros2;
cout << "\nResultado de resultadoSuma + enteros1 + enteros2 " <<
                                        resultadoFinal << endl;
.....................................................
```

Sobrecarga del operador +

La salida obtenida es:

```
................................................
Lectura de array1 con tam.: 7
1 2 3 4 5 6 7

Lectura de array2 con tam.: 10
8 9 10 11 12 13 14 15 16 17

Resultado de la suma de enteros1 y enteros2

1 2 3 4 5 6 7 8 9 10 11 12 13 14 15 16 17

Resultado de resultadoSuma + enteros1 + enteros2

1 2 3 4 5 6 7 8 9 10 11 12 13 14 15 16 17 1 2 3 4 5 6 7 8 9 10 11 12 13 14 15 16 17
................................................
```

Sobrecarga del operador + I

También puede interesar disponer de una operación que permita agregar un elemento único a un array. Haciendo uso de ella, podríamos hacer:

```
4 + array1; // Agregaria 4 como primer elemento
array2 + 5  // Agregaria 5 como ultimo elemento
```

Se deja como ejercicio pensar cómo habría que implementar la sobrecarga del operador + para tratar este caso. Para la primera sentencia, ¿es posible la implementación como método de la clase?

Índice

Sobrecarga del operador $+=$

Al usar el operador $+=$ sobre arrays deseamos agregar contenido a un array (el contenido del array de la derecha se agregará al de la izquierda):

```
array1 += array2; // array1 y array2 son objetos....
```

Al ser ambos operandos objetos de la clase podemos optar por las dos alternativas ya vistas. Usaremos la implementación mediante método de la clase.

Sobrecarga del operador $+=$

La declaración del método se hace en la parte pública del archivo de cabecera de la clase:

```
.....................................................................
   /**
    * Sobrecarga del operador compuesto de asignacion e
    * incremento
    * @param otro
    * @return
    */
   const Array & operator+=(const Array &otro);
.....................................................................
```

Sobrecarga del operador $+=$

Observaciones:

- el método devuelve una referencia constante para evitar que pueda aplicarse el operador de forma repetida: $(a+=b)+=c$;
- el método devuelve el objeto apuntado por **this**, ya que ha sido modificado por la operación.

Sobrecarga del operador $+=$ I

La implementación es:

```
/**
 * Sobrecarga del operador compuesto de asignacion e
 * incremento
 * @param otro
 * @return
 */
const Array & Array::operator+=(const Array &otro){
   // Se reserva espacio para otro array de valores
   int *ptrNuevo=new int[size+otro.size];

   // Se copian los elementos de this->ptr sobre el nuevo array
   for(int i=0; i < size; i++){
      ptrNuevo[i]=ptr[i];
   }

   // Ahora ya puede liberarse ptr
   delete [] ptr;
```

Sobrecarga del operador $+=$ II

```
    // Se copian los elementos del segundo sumando
    for(int i=0; i < otro.size; i++){
       ptrNuevo[size+i]=otro.ptr[i];
    }

    // Ahora hay que hacer que ptr apunte a ptrNuevo
    ptr=ptrNuevo;

    // Se cambia el valor de size
    size+=otro.size;

    // Se devuelve el propio objeto modificado
    return *this;
}
```

Sobrecarga del operador $+=$ I

Y probamos si funciona bien con un ejemplo:

```
.....................................................
// Recordemos que enteros1 tiene tam. 7 y enteros2 10
// Se prueba la lectura sobre el array de 7 elementos
cout << "\nLectura de array1 con tam.: " << enteros1.getSize() << endl;
cin >> enteros1;

// Se prueba la lectura sobre el array de 10 elementos
cout << "\nLectura de array2 con tam.: " << enteros2.getSize() << endl;
cin >> enteros2;

// Se prueba el operador recien creado
enteros1 += enteros2;

// Se muestra el resultado
cout << "Resultado de la suma de enteros1 y enteros2 " << endl;
cout << enteros1 << endl;
.....................................................
```

Sobrecarga del operador $+=$

La salida obtenida es:

```
.................................................
Lectura de array1 con tam.: 7
1 2 3 4 5 6 7

Lectura de array2 con tam.: 10
8 9 10 11 12 13 14 15 16 17
Resultado de la suma de enteros1 y enteros2

1 2 3 4 5 6 7 8 9 10 11 12 13 14 15 16 17
.................................................
```

Sobrecarga del operador $+=$ I

Al igual que en el caso anterior, podría interesar disponer de la posibilidad de ejecutar operaciones del estilo:

```
array1+=4;
```

que agrega un elemento a **array1**, al final. ¿Cómo podría implementarse esta funcionalidad? Siempre hemos de considerar qué tipo está asociado al operando a la izquierda del operador.

Índice

Sobrecarga del operador ==

El primero de los operadores relacionales considerados es el de igualdad: devolverá verdadero si tienen el mismo contenido y falso en caso contrario.

```
array1 == array2; // array1 y array2 son objetos....
```

Al ser ambos operandos objetos de la clase podemos optar por las dos alternativas ya vistas. Usaremos la implementación mediante método de la clase.

Sobrecarga del operador ==

La declaración del método se hace en la parte pública del archivo de cabecera de la clase:

```
.....................................................................
   /**
    * Sobrescritura del operador de igualdad
    * @param otro
    * @return resultado de la comparacion
    */
   bool operator==(const Array &otro) const;
.....................................................................
```

Sobrecarga del operador ==

Observaciones:

- el método devuelve un valor booleano
- si se usa el mismo objeto a ambos lados del operador el resultado debe ser verdadero
- si se comparan dos objetos diferentes, deben coincidir sus contenidos

Sobrecarga del operador == I

La implementación es:

```
/**
 * Sobrescritura del operador de igualdad
 * @param otro
 * @return resultado de la comparacion
 */
bool Array::operator==(const Array &otro) const{
   bool resultado=false;

   // Si se trata del mismo objeto, no hace falta
   // mirar nada mas
   if (this == &otro){
      resultado=true;
   }
   else{
      // Se comprueba ahora el tamaño. Solo hay que
      // comparar valores si ambos arrays tienen el
      // mismo tamaño
      if (size == otro.size){
         resultado=true;
```

Sobrecarga del operador == II

```
            for(int i=0; i < size && resultado; i++){
               if (ptr[i] != otro.ptr[i]){
                  resultado=false;
               }
            }
         }
      }

      // Se devuelve el valor de resultado
      return resultado;
   }
```

Sobrecarga del operador = I

Y probamos si funciona bien con un ejemplo:

```
.....................................................
// Recordemos que enteros1 tiene tam. 7 y enteros2 10
// Se hace la lectura sobre el array de 10 elementos
cout << "\nLectura de array2 con tam.: " << enteros2.getSize() << endl;
cin >> enteros2;

// Se prueba el operador recien creado
enteros1 = enteros2;

// Se muestra el resultado
cout << "enteros1: " << enteros1 << endl;
cout << "enteros2: " << enteros2 << endl;
cout << "Res. de enteros1 == enteros2 : " << (enteros1 == enteros2) << endl;
.....................................................
```

Sobrecarga del operador ==

La salida obtenida es:

```
................................................
Lectura de array2 con tam.: 10
1 2 3 4 5 6 7 8 9 10
enteros1:
1 2 3 4 5 6 7 8 9 10

enteros2:
1 2 3 4 5 6 7 8 9 10

Res. de enteros1 == enteros2 : 1
................................................
```

¿Cómo se sobrescribiría el operador != a partir de esta implementación?

Índice

Sobrecarga del operador []

La implementación de este operador requiere considerar dos situaciones diferentes: la operación devuelve una dirección en que poder almacenar un valor (**l-value**):

```
array1[1] = 3; // array1 es un objeto...
```

También se usará en situaciones en que el operador devuelve únicamente un valor (**r-value**). La aplicación de este operador está pensada para arrays constantes cuyos valores no deben modificarse. En este caso siempre se usará esta versión del operador que devuelve sólo valores:

```
int x = array1[1]; // array1 es un array constante
```

¿Y si en el caso anterior el array no fuera constante?

Sobrecarga del operador []

La declaración de ambos métodos se hace en la parte pública del archivo de cabecera de la clase:

```
.....................................................................
   /**
    * Sobrescritura del operador corchete, para que pueda
    * comportarse como lvalue y asignar valor sobre el
    * @param
    * @return
    */
   int & operator[](int);

   /**
    * Sobrecarga del operador [], para que pueda comportarse
    * como rvalue
    * @param
    * @return
    */
   int operator[](int) const;
.....................................................................
```

Estas dos situaciones precisan implementaciones diferentes.

Sobrecarga del operador []

Observaciones:

- en el primer caso el método debe devolver una referencia a una posición del array
- en el segundo devuelve un valor y el método debe declararse como constante, ya que no se modificará su contenido
- la aplicación de uno u otro dependerá de si el objeto sobre el que se hace la llamada es constante o no. Si es constante se usa la segunda versión

Sobrecarga del operador []

La implementación es:

```
/**
 * Sobrescritura del operador corchete, para que pueda
 * comportarse como lvalue y asignar valor sobre el
 * @param indice a acceder
 * @return
 */
int & Array::operator[](int indice){
   // Se comprueba que el indice pasado como argumento sea valido
   if (indice < 0 || indice >= size){
      // Se genera una excepcion
      throw out_of_range("Indice fuera de rango en acceso con []");
   }

   // Si todo ha ido bien, devuelve el valor
   return ptr[indice];
}
```

Sobrecarga del operador []

```
/**
 * Sobrescritura del operador corchete, para que pueda
 * comportarse como rvalue
 * @param indice a acceder
 * @return
 */
int Array::operator[](int indice) const{
   // Se comprueba que el indice pasado como argumento sea valido
   if (indice < 0 || indice >= size){
      // Se genera una excepcion
      throw out_of_range("Indice fuera de rango en acceso con []");
   }

   // Si todo ha ido bien, devuelve el valor
   return ptr[indice];
}
```

Sobrecarga del operador [] I

Y probamos si funciona bien con un ejemplo:

```
.....................................................
// Se prueba la lectura sobre el array de 10 elementos
cout << "\nLectura de array2 con tam.: " << enteros2.getSize() << endl;
cin >> enteros2;

// Se prueba el operador recien creado como l-value
enteros2[0]=58;

// Se muestra el resultado
cout << "enteros2: " << enteros2 << endl;

// Se usa como r-value
const Array enteros3=enteros2:
cout << "Elemento en posicion 3: " << enteros3[3] << endl;
.....................................................
```

Sobrecarga del operador []

La salida obtenida es:

```
.................................................
Lectura de array2 con tam.: 10
1 2 3 4 5 6 7 8 9 10
enteros2:
58 2 3 4 5 6 7 8 9 10  // version para l-value

Elemento en posicion 3: 4 // version para r-value
.................................................
```

Índice

La clase mínima I

Recomendaciones:

- sólo se incluyen aquellos constructores y métodos que sean estrictamente necesarios
- habitualmente suele necesitarse el constructor por defecto
- cuando la clase tiene datos miembro que usan memoria dinámica, añadiremos: **constructor de copia**, **destructor** y **operador de asignación**.

```
class Array {
    private:
        int *ptr;
        int size;
    public:
        Array();     // Constructor por defecto
        Array(const Array &p); // Constructor de copia
        ~Array();    // Destructor
        Array& operator=(const Array &objeto);
```

La clase mínima II

```
        ..................................
};
```

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