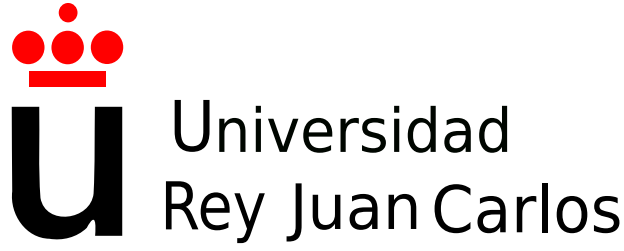
5. Sobrecarga de operadores

Julio Vega

julio.vega@urjc.es



Contenidos

- Comunicación entre objetos mediante sus funciones es incómoda.
  - E.g. clase `Matemática`, pensemos en las llamadas a sus funciones...
  - La sobrecarga logra un código más claro que con llamadas a funciones.
- Muchas manipulaciones comunes se realizan con operadores (<<).
  - Sobrecargar operadores es hacer que estos puedan manipular objetos.
- *La sobrecarga es la polisemia de los operadores.*
  - Polisemia: cuando una misma palabra tiene varios significados.
  - E.g. Banco: dinero, asiento. <<: op. flujo, *shift* bits.
- C++ no permite crear, pero sí sobrecargar muchos de sus operadores.
  - El compilador genera el código apropiado según el contexto.

Operadores que se pueden sobrecargar

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

Operadores que no se pueden sobrecargar

```
. .* :: ?:
```

- Mediante método no `static` o con función global.
- Nombre de función: `operator` seguida del símbolo a sobrecargar.

Def. `Phone.h` con operadores flujo sobrecargados como func. friend

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

using namespace std;

class Phone {
  friend ostream &operator<< (ostream &, const Phone &);
  friend istream &operator>> (istream &, Phone &);
private:
  string countryCode; // 2-digit country code (e.g. "34")
  string areaCode; // 3-digit area code (e.g. "924")
  string number; // 6-digit number
}; // end class Phone
```

Implementación de la clase Phone (Phone.cpp)

```
#include <iomanip>
#include "Phone.h"
using namespace std;
ostream &operator<< (ostream &output, const Phone &phone) {
  output << "(" << phone.countryCode << ") "
    << phone.areaCode << "-" << phone.number;
  return output; // enables cout << a << b << c;
} // end function operator<< [e.g. format: (+34) 924-736515]

istream &operator>> (istream &input, Phone &phone) {
  input.ignore (2); // skip ( and +
  input >> setw (2) >> phone.countryCode; // input country code
  input.ignore (2); // skip ) and space
  input >> setw (3) >> phone.areaCode; // input area code
  input.ignore (); // skip dash (-)
  input >> setw (6) >> phone.number; // input number
  return input; // enables cin >> a >> b >> c;
} // end function operator>>
```

Ejemplo de uso de la clase Phone, con operadores sobrecargados

```
#include <iostream>
#include "Phone.h"
using namespace std;

int main() {
  Phone phone; // create object phone
  cout << "Enter phone number (format (+34) 924-736515):\n";

  // cin >> phone invokes operator>> by implicitly issuing
  // the global function call operator>> (cin, phone)
  cin >> phone; // >> overloaded!

  cout << "The phone number entered was: ";

  // cout << phone invokes operator<< by implicitly issuing
  // the global function call operator<< (cout, phone)
  cout << phone << endl; // << overloaded!
} // end main
```

- (A) Método no `static` (sin args) o (B) func. global (con un arg.).

(A) Def. `Cadena.h` con operador ! sobrecargado como método

```
class Cadena { // String could be confused with C++ class
  bool operator!() const; // return if string is empty
}; // end class Cadena
```

(B) Declaraciones operador ! sobrecargado como funciones globales

```
bool operator! (const Cadena); // Op. 1: with an object as arg.
bool operator! (const Cadena &); // Op. 2: ref. to object as arg.
```

- Op. 1: se crea copia de obj.; la función no altera obj. original.
- Op. 2: no se hace copia de obj.; la función altera obj. original.

Uso del operador ! sobrecargado para devolver si cadena vacía

```
Cadena miCadena;
if (!miCadena) cout << "Empty string" << endl;
```

- Todo lo ya visto es aplicable a este tipo de operadores.
- (A) Método no `static` (un arg.) o (B) func. global (dos args).

(A) Def. `Cadena.h` con operador $<$ sobrecargado como método

```
class Cadena { // String could be confused with C++ class
  bool operator<() const; // return if left string is smaller
}; // end class Cadena
```

(B) Declaraciones operador $<$ sobrecargado como función global

```
bool operator< (const Cadena &, const Cadena &);
// could it be defined with two objects as args (as seen above)
```

- Los programas procesan información de muchos tipos.
- A veces, todas las operaciones *permanecen* dentro de un tipo.
  - E.g. al sumar un `int` con un `int` se produce un `int`.
- Pero muchas veces se necesita convertir datos de un tipo a otro.
  - E.g. cálculos, paso de valores a funciones, devolución de valores.
- Podemos usar operadores de conversión de tipos para forzarlas:

```
int main () {
   char c;
   int i = 34;
   float f = 1.5;
   double d;

   c = static_cast<char>(i); // int to char
   d = static_cast<double>(f); // float to double
}
```

- Pero, ¿y los tipos definidos por el usuario? El compilador no sabe.

- Para ello son necesarios los **constructores de conversión**.
  - Son const. con un arg. que convierte objetos de un tipo a otro.
  - `A::operator int () const; // obj. tipo A -> tipo int`
    - Esta función convierte obj. `A` (de usuario) en tipo `int`.
  - Y así, cuando usemos el operador `cast` de conversión...
    - `static_cast<int> (a) // Habiendo declarado A a;`
  - ...el compilador genera la llamada:
    - `a.operator int()`
- Estas func. deben ser un método no `static` (de la clase A).
- Y deben ser `const` porque no modifican el objeto original.
- Se pueden definir varias funciones de operador de conversión.
  - `A::operator char* () const; // obj. tipo A -> tipo char*`
  - `A::operator B () const; // obj. tipo A -> tipo B`

- Tanto en prefijo como postfijo, se pueden sobrecargar.
- Las funciones son distintas, así el compilador distingue versiones:

```
A &operator++ (); // pre-increment (++a) as a class method
A &operator++ (A &); // pre-increment (++a) as a global function
A operator++ (int); // post-increment (a++) as a class method
A operator++ (A &, int); // post-incr. (a++) as a global function
```

- Todo lo anterior es igualmente aplicable al decremento.

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

int main() {
  string s1 ("Hello");
  string s2 (" world!");
  string s3;

  cout << "s1 = \"" << s1 << "\"; s2 = \"" << s2
    << "\n(s1 == s2): " << (s1 == s2 ? "true" : "false")
    << "\n(s1 < s2): " << (s1 < s2 ? "true" : "false")
    << "\n(s1 >= s2): " << (s2 >= s1 ? "true" : "false");

  if (s3.empty()) {
    s1 += s2;
    s3 = s1;
  }
  cout << "\ns3 = \"" << s3 << endl;
}
```

5. Sobrecarga de operadores

Julio Vega

julio.vega@urjc.es