Costruttore di Conversione

Conversione tra tipi diversi

- In generale le operazioni si svolgono fra dati con lo stesso tipo (es. somme fra interi che restituiscono interi)
- può però sorgere la necessità di convertire dati da un tipo all'altro

Conversione tra tipi diversi

- un caso tipico è quello appena presentato in cui sarebbe necessario ridefinire esplicitamente ogni operatore per le varie combinazioni di tipi in ingresso:
  - `operator+(int, MyInt obj1);`
  - `operator+(MyInt obj1, int);`
  - `operator+(MyInt obj1, MyInt obj2);`
- una alternativa è quella di utilizzare delle conversioni
- tramite la conversione si avrebbe:
  - `3+o; //operator+(MyInt(int), MyInt obj1);`
  - `o+3; //operator+(MyInt obj1, MyInt(int));`
  - `o1+o2; //operator+(MyInt obj1, MyInt obj2);`

Costruttore di conversione

- Per implementare la conversione si utilizza semplicemente un costruttore che prende in ingresso il tipo desiderato
- tale costruttore viene indicato con il nome di *costruttore di conversione*
- la sintassi è quella ordinaria di un costruttore:

```
NomeClasse(tipo var){//definizione}
```

Esempio

```
#ifndef HUGEINT1_H
#define HUGEINT1_H
#include <iostream>
class HugeInt {
   friend ostream &operator<<( ostream &, const HugeInt & );
public:
   HugeInt( long = 0 );       // conversion/default constructor
   HugeInt( const char * );    // conversion constructor
   HugeInt operator+( const HugeInt & ); // add another HugeInt
   HugeInt operator+( int );             // add an int
   HugeInt operator+( const char * ); // add an int in a char *
private:
   short integer[ 30 ];
};
#endif
```

```
#include <cstring>
#include "hugeint1.h"
// Conversion constructor
HugeInt::HugeInt( long val )
{
   int i;
   for ( i = 0; i <= 29; i++ )
      integer[ i ] = 0;   // initialize array to zero
   for ( i = 29; val != 0 && i >= 0; i-- ) {
      integer[ i ] = val % 10;
      val /= 10;
   }
}
```

```
HugeInt::HugeInt( const char *string )
{
   int i, j;
   for ( i = 0; i <= 29; i++ )
      integer[ i ] = 0;
   for ( i = 30 - strlen( string ), j = 0; i <= 29; i++, j++ )
      if ( isdigit( string[ j ] ) )
         integer[ i ] = string[ j ] - '0';
}
```

```
// Addition
HugeInt HugeInt::operator+( const HugeInt &op2 )
{
   HugeInt temp;
   int carry = 0;

   for ( int i = 29; i >= 0; i-- ) {
      temp.integer[ i ] = integer[ i ] +
                          op2.integer[ i ] + carry;

      if ( temp.integer[ i ] > 9 ) {
         temp.integer[ i ] %= 10;
         carry = 1;
      }
      else carry = 0;
   }
return temp;
}
```

```
// Addition
HugeInt HugeInt::operator+( int op2 )
   { return *this + HugeInt( op2 ); }

// Addition
HugeInt HugeInt::operator+( const char *op2 )
   { return *this + HugeInt( op2 ); }

ostream& operator<<( ostream &output, const HugeInt &num )
{
   int i;
   for ( i = 0; ( num.integer[ i ] == 0 ) && ( i <= 29 ); i++ );
   if ( i == 30 ) output << 0;
   else
      for ( ; i <= 29; i++ )
         output << num.integer[ i ];
   return output;
}
```

Esempio

```
// Test driver for HugeInt class
#include <iostream>
#include "hugeint1.h"
int main()
{
   HugeInt n1( 7654321 ), n2( 7891234 ),
           n3( "99999999999999999999999999999" ),
           n4( "1" ), n5;
   cout << "n1 is " << n1 << "\nn2 is " << n2
        << "\nn3 is " << n3 << "\nn4 is " << n4
        << "\nn5 is " << n5 << "\n\n";
   n5 = n1 + n2;
   cout << n1 << " + " << n2 << " = " << n5 << "\n\n";
   cout << n3 << " + " << n4 << "\n= " << ( n3 + n4 )
        << "\n\n";
   n5 = n1 + 9;
   cout << n1 << " + " << 9 << " = " << n5 << "\n\n";
   n5 = n2 + "10000";
   cout << n2 << " + " << "10000" << " = " << n5 <<
  endl;
   return 0;
}
```

Esercizio

- Scrivere la definizione degli operatori di confronto per la classe HugeInt

Operatore di conversione

- Un costruttore di conversione non può però
  - specificare una conversione da un tipo utente ad un tipo base (i tipi base non sono oggetti e non se ne possono quindi sovraccaricare i metodi)
  - specificare una conversione da un tipo utente A ad un altro tipo utente B senza modificare B (potremmo non avere accesso alla classe B)
- per questi casi si utilizza un *operatore di conversione*
- Sintassi:

```
Classe::operator tipo() const{ //definizione }
```

- Nota: non si indica il tipo di ritorno perché è ovvio dalla conversione
- Esempio:

```
HugeInt::operator int() const{ //definizione }
```

Note su l'operatore di conversione

- attenzione quando il passaggio da un tipo ad un altro implica perdita di informazione
  - Es da HugeInt a int
- attenzione alle ambiguità quando si usano operatori predefiniti
  - Es: HugeInt a; int b; a+b
  - quale tra?:
    - operator+(a,HugeInt(b))
    - int(a)+b
- è bene avere uno solo dei due meccanismi ma non entrambi

Overloading degli operatori >> e <<

- Operatori utili sono quelli di inserzione ed estrazione dallo stream
- vanno ridefiniti come funzioni friend per poterli usare come cin>>obj e cout<<obj
- l'istruzione cin>>obj (cout<<obj) invoca la chiamata di funzione operator>>(cin, obj) (operator<<(cout,obj))

Esempio

```
#include <iostream>
#include <iomanip>
class PhoneNumber {
   friend ostream &operator<<( ostream&, const PhoneNumber & );
   friend istream &operator>>( istream&, PhoneNumber & );
private:
   char areaCode[ 4 ];  // 3-digit area code and null
   char exchange[ 4 ];  // 3-digit exchange and null
   char line[ 5 ];      // 4-digit line and null
};
// Overloaded stream-insertion operator (cannot be
// a member function if we would like to invoke it with
// cout << somePhoneNumber;).
ostream &operator<<( ostream &output, const PhoneNumber &num )
{
   output << "(" << num.areaCode << ") "
          << num.exchange << "-" << num.line;
   return output;     // enables cout << a << b << c;
}
```

```
istream &operator>>( istream &input, PhoneNumber &num )
{
   input.ignore();                          // skip (
   input >> setw( 4 ) >> num.areaCode; // input area code
   input.ignore( 2 );                       // skip ) and space
   input >> setw( 4 ) >> num.exchange; // input exchange
   input.ignore();                          // skip dash (-)
   input >> setw( 5 ) >> num.line;     // input line
   return input;       // enables cin >> a >> b >> c;
}
int main()
{
   PhoneNumber phone; // create object phone
   cout << "Enter phone number in the form (123) 456-
  7890:\n";
   // cin >> phone invokes operator>> function by
   // issuing the call operator>>( cin, phone ).
   cin >> phone;
   // cout << phone invokes operator<< function by
   // issuing the call operator<<( cout, phone ).
   cout << "The phone number entered was: " << phone <<
  endl;
   return 0;
}
```

Esempio: classe array

- Realizziamo una classe per vettori di interi
- le specifiche includono:
  - controllo sugli indici
  - assegnazione unitaria
  - operatori di egualianza, diversità unitari
  - stampa e input unitari tramite << e >>

```
// Simple class Array (for integers)
#ifndef ARRAY1_H
#define ARRAY1_H
#include <iostream>
class Array {
   friend ostream &operator<<( ostream &, const Array & );
   friend istream &operator>>( istream &, Array & );
public:
   Array( int = 10 );                          // default constructor
   Array( const Array & );                     // copy constructor
   ~Array();                                   // destructor
   int getSize() const;                        // return size
   const Array &operator=( const Array & ); // assign arrays
   bool operator==( const Array & ) const;  // compare equal

   // Determine if two arrays are not equal and
   // return true, otherwise return false (uses operator==).
   bool operator!=( const Array &right ) const
      { return ! ( *this == right ); }

   int &operator[]( int );                     // subscript operator
   const int &operator[]( int ) const;  // subscript operator
   static int getArrayCount();                 // Return count of
                                               // arrays
   instantiated.
private:
   int size; // size of the array
   int *ptr; // pointer to first element of array
   static int arrayCount;  // # of Arrays instantiated
};
#endif
```

```
//definitions
#include <iostream>
using std::cout;
using std::cin;
using std::endl;

#include <iomanip>
using std::setw;

#include <cstdlib>
#include <cassert>
#include "array1.h"

// Initialize static data member at file scope
int Array::arrayCount = 0;   // no objects yet

// Default constructor for class Array (default size
   10)
Array::Array( int arraySize )
{
   size = ( arraySize > 0 ? arraySize : 10 );
   ptr = new int[ size ]; // create space for array
   assert( ptr != 0 );    // terminate if memory not
   allocated
   ++arrayCount;          // count one more object

   for ( int i = 0; i < size; i++ )
      ptr[ i ] = 0;           // initialize array
}
```

```
// Copy constructor for class Array
// must receive a reference to prevent infinite
   recursion
Array::Array( const Array &init ) : size( init.size )
{
   ptr = new int[ size ]; // create space for array
   assert( ptr != 0 );    // terminate if memory not
   allocated
   ++arrayCount;          // count one more object

   for ( int i = 0; i < size; i++ )
      ptr[ i ] = init.ptr[ i ];  // copy init into
   object
}

// Destructor for class Array
Array::~Array()
{
   delete [] ptr;             // reclaim space for array
   --arrayCount;              // one fewer objects
}

// Get the size of the array
int Array::getSize() const { return size; }
```

```
// Overloaded assignment operator
const Array &Array::operator=( const Array &right )
{
   if ( &right != this ) {  // check for self-
   assignment

      // for arrays of different sizes, deallocate
   original
      // left side array, then allocate new left side
   array.
      if ( size != right.size ) {
         delete [] ptr;          // reclaim space
         size = right.size;      // resize this object
         ptr = new int[ size ]; // create space for
   array copy
         assert( ptr != 0 );     // terminate if not
   allocated
      }

      for ( int i = 0; i < size; i++ )
         ptr[ i ] = right.ptr[ i ];  // copy array into
   object
   }

   return *this;   // enables x = y = z;
}
```

```
// Determine if two arrays are equal and
// return true, otherwise return false.
bool Array::operator==( const Array &right ) const
{
   if ( size != right.size )
      return false;    // arrays of different sizes

   for ( int i = 0; i < size; i++ )
      if ( ptr[ i ] != right.ptr[ i ] )
         return false; // arrays are not equal

   return true;         // arrays are equal
}

// Overloaded subscript operator for non-const Arrays
// reference return creates an lvalue
int &Array::operator[]( int subscript )
{
   // check for subscript out of range error
   assert( 0 <= subscript && subscript < size );

   return ptr[ subscript ]; // reference return
}
```

```
// Overloaded subscript operator for const Arrays
// const reference return creates an rvalue
const int &Array::operator[]( int subscript ) const
{
   // check for subscript out of range error
   assert( 0 <= subscript && subscript < size );

   return ptr[ subscript ]; // const reference return
}

// Return the number of Array objects instantiated
// static functions cannot be const
int Array::getArrayCount() { return arrayCount; }
```

```
// Overloaded input operator for class Array;
// inputs values for entire array.
istream &operator>>( istream &input, Array &a )
{
   for ( int i = 0; i < a.size; i++ )
      input >> a.ptr[ i ];

   return input;   // enables cin >> x >> y;
}

// Overloaded output operator for class Array
ostream &operator<<( ostream &output, const Array &a )
{
   int i;

   for ( i = 0; i < a.size; i++ ) {
      output << setw( 12 ) << a.ptr[ i ];

      if ( ( i + 1 ) % 4 == 0 ) // 4 numbers per row of
   output
         output << endl;
   }

   if ( i % 4 != 0 )
      output << endl;

   return output;   // enables cout << x << y;
}
```

```
// Clientfor simple class Array
#include <iostream>
#include "array1.h"

int main()
{
   // no objects yet
   cout << "# of arrays instantiated = "
        << Array::getArrayCount() << '\n';//out:# of
  arrays instantiated = 0

   // create two arrays and print Array count
   Array integers1( 7 ), integers2;
   cout << "# of arrays instantiated = "
        << Array::getArrayCount() << "\n\n"; //out:# of
  arrays instantiated = 2

   // print integers1 size and contents
   cout << "Size of array integers1 is "
        << integers1.getSize()
        << "\nArray after initialization:\n"
        << integers1 << '\n';
/* Size of array integers1 is 7
Array after initialization:
0      0      0      0
0      0      0
*/
```

```
// print integers2 size and contents
 cout << "Size of array integers2 is "
      << integers2.getSize()
      << "\nArray after initialization:\n"
      << integers2 << '\n';

 // input and print integers1 and integers2
 cout << "Input 17 integers:\n";
 cin >> integers1 >> integers2;
 cout << "After input, the arrays contain:\n"
      << "integers1:\n" << integers1
      << "integers2:\n" << integers2 << '\n';

 // use overloaded inequality (!=) operator
 cout << "Evaluating: integers1 != integers2\n";
 if ( integers1 != integers2 )
    cout << "They are not equal\n";

 // create array integers3 using integers1 as an
 // initializer; print size and contents
 Array integers3( integers1 );

 cout << "\nSize of array integers3 is "
      << integers3.getSize()
      << "\nArray after initialization:\n"
      << integers3 << '\n';
```

```
   // use overloaded assignment (=) operator
    cout << "Assigning integers2 to integers1:\n";
    integers1 = integers2;
    cout << "integers1:\n" << integers1
         << "integers2:\n" << integers2 << '\n';

    // use overloaded equality (==) operator
    cout << "Evaluating: integers1 == integers2\n";
    if ( integers1 == integers2 )
       cout << "They are equal\n\n";

    // use overloaded subscript operator to create
   rvalue
    cout << "integers1[5] is " << integers1[ 5 ] <<
   '\n';

    // use overloaded subscript operator to create
   lvalue
    cout << "Assigning 1000 to integers1[5]\n";
    integers1[ 5 ] = 1000;
    cout << "integers1:\n" << integers1 << '\n';

    // attempt to use out of range subscript
    cout << "Attempt to assign 1000 to integers1[15]" <<
   endl;
    integers1[ 15 ] = 1000;  // ERROR: out of range

    return 0;
```

Nota

- Si ricorda che:
  - mentre per tutti gli operatori l'associatività è da sinistra a destra
  - l'associatività dell'operatore = è da destra a sinistra
- pertanto
  - x+y+z viene valutata come (x+y)+z
- mentre
  - a=b=c viene valutata come a=(b=c)

Nota

- Esistono due versioni dell'operatore [ ], una const e una non const
- quella const è utilizzata quando stiamo operando su un array dichiarato const
- passare ad una funzione un array const e poi essere in grado di modificare il contenuto dell'array accedendo tramite l'operatore [ ] ai valori del vettore sarebbe un errore logico
- il compilatore pertanto non permette l'uso di funzioni membro non const con oggetti const