# Elementi di C++ avanzato

Luca Lista

*INFN, Sezione di Napoli*

# Argomenti trattati:

- Conversione di tipi (Cast)
- Run-Time Type Information (RTTI)
- Namespace
- Gestione delle eccezioni
- Input/Output con gli Stream

# Cast

- Il *cast* consente di cambiare il tipo di un oggetto
- E' un'operazione molto rischiosa e da fare solo in casi veramente eccezionali
- L'uso dei cast può in casi molto limitati essere indispensabile…
  - Esempio: per accedere direttamente a registri di una scheda di un sistema di acquisizione
- …ma tradizionalmente se ne fa abuso
- Rispetto al C, il C++ fornisce strumenti per effettuare il cast molto più sofisticati

# Reinterpret e Static cast

- Un indirizzo qualsiasi di memoria può essere reinterpretato come puntatore a un oggetto di un tipo particolare

```
IOdev * dev = reinterpret_cast<IOdev*>( 0xff00 );
```

- Equivale al cast C: `dev = (Iodev*)( 0xff00 );`
- Può essere usato in stile C per trasformare puntatori `void*` (Deprecato!).
- Qualche controllo in più a livello di compilatore viene fatta con `static_cast<…>`, che offre maggiori sicurezze (conversione nella stessa gerarchia di classi).
- Attenzione! Estremamente rischioso!

# Const cast

- Permette di violare l'accesso costante ad un oggetto:

```
Correntista tizio;
const contoBancario& conto = banca.conto(tizio);
contoBancario & c = const_cast<contoBancario&> (conto);
c.preleva(10000000);
```

- Attenzione: anche qui se un oggetto è dichiarato **`const`** ce ne sarà un motivo.
- E' anche possibile modificare lo stato di un oggetto in un metodo selettore (!!!)

# Dynamic cast

- Se ho un albero di ereditarietà il dynamic cast consente di risalire a quale sottotipo appartenga un oggetto noto come tipo astratto

```
Shape * s = something;
Circle * c = dynamic_cast<Circle*>( s );
if ( c != 0 ) r = c->radius();
```

- Se l'oggetto non è del tipo richiesto, il dynamic cast restituisce 0 (zero)

# RTTI

- Per conoscere ulteriori informazioni su un tipo particolare si può usare il Run-Time Type Information

```
#include <typeinfo>

Shape * s = something;
const type_info& rtti = typeid( *s );
cout << rtti.name() << endl;
if (rtti == typeid(Circle) ) // . . .
```

# Namespace

- E’ possibile separare i nomi di diverse classi in ambiti separati (esempio: due pacchetti che hanno concetti diversi per lo stesso nome, esempio: **`Punto`**)

**Geometry/Punto.h**

```
namespace Geometry
{
  class Punto
  {
     // . . .
  };
}
```

```
#include “Geometry/Punto.h”

using namespace Geometry;

void main()
{
  Punto p1, p2;
}
```

# Gestione delle eccezioni (1)

- E’ anche possibile causare un’interruzione dovuta ad errore

```
if ( denominator == 0 ) throw DivideByZero;

if ( inputLine.invalid() ) throw SyntaxError(“invalid command”);
```

- Dove in precedenza abbiamo definito gli *handler* come:

```
class DivideByZero {};

class SyntaxError
{
public:
  SyntaxError( const char* c ) : c_( c );
  const string& error() const { return c_; }
private:
  string c_;
};
```

# Gestione delle eccezioni (2)

- L'errore può essere gestito dal programma in modo che non venga interrotta l'esecuzione:

```
try { myCode(); }
catch( DivideByZero )
{
  cerr << "Attento: hai tentato"
       << "una divisione per zero!" << endl;
}
catch( SyntaxError s )
{
  cerr << "Errore di sintassi: " s.error() << endl;
}
catch( . . . )
{
  cerr << "Errore sconosciuto" << endl;
}
```

# Gestione dell'I/O: gli stream

- Diversi stream (**`cout`**, **`cin`**, **`cerr`**, etc.) hanno un'interfaccia comune che permette la definizione di molte operazioni in comune per diversi stream
- Già abbiamo visto gli operatori:

```
ostream& << (ostream&, const T&);
istream& >> (istream&, const T&);
```

- Che permettono di scrivere:

```
T x, y;
cout << x;
cin >> y;
```

# Altri modi di usare l'I/O (1)

- Esistono altri modi per accedere agli stream:
- Leggere un carattere alla volta:

```
char c;
cin.get(c)
```

```
char c1, c2;
cin.get(c1).get(c2);
```

- Scrivere un carattere alla volta:

```
char c;
cout.put(c);
```

```
char c1, c2;
cout.put(c1).put(c2) << endl;
```

# Altri modi di usare l'I/O (2)

- Leggere un buffer:

```
char c[256];
cin.get( c, 256, '\n' ); // \n = terminatore
```

  - Il terminatore si può anche omettere se è `'\n'`
  - il terminatore viene lasciato come prossimo carattere da leggere

```
cin.getline( c, 256 ); // '\n' sottinteso
```

  - il terminatore viene rimosso, così la prossima operazione non fallisce

- Ignorare una parte dello stream

```
cin.ignore( 256 );
```

  - Di default `cin.ignore()` rimuove un solo carattere.

# Formato dell'output

```
#include <iostream>
#include <iomanip>

void  main()
{
        // hexadecimal :          0x1000
  cout.setf(ios::hex, ios::basefield); cout.setf(ios::showbase);
  cout << 4096 << endl;

        // octal:                 0100000
  cout.setf(ios::oct, ios::basefield); cout.setf(ios::showbase);
  cout << 4096 << endl;

        // floating point         ########3.1415000000
  cout.setf(ios::fixed, ios::floatfield);
  cout << setw(20) << setfill('#') << setprecision(10) << 3.1415 << endl;

        // scientific exponential ####3.1415000000e+00
  cout.setf(ios::scientific, ios::floatfield);
  cout << setw(20) << setfill('#') << setprecision(10) << 3.1415 << endl;

        // reset to default       3.1415
  cout.setf(ios::fmtflags(0), ios::floatfield);
  cout << 3.1415 <<endl;
}
```

# Stati di uno stream

- Alcune funzioni permettono di investigare lo stato di uno stream:

```
bool good() const;    // OK prossimo I/O
bool bad() const;     // stream corrotto
bool eof() const;     // visto End Of File
bool fail() const;    // KO prossimo I/O
operator void*() const; // !fail()
operator !() const;     // fail()
```

- Esempi:

```
while( ! cin.eof() )
{
  cin >> c;
}
```

```
char c;
while( cin >> c )
  cout << c;
```

# Input e Output su file

```
#include <fstream>

int main()
{
  char* filename = "test-out.txt";
  ofstream fout ( filename );
  if ( ! fout ) { cerr << " can't open input - " << filename <<
     endl;  return 1; }

  fout << 3.1415 << endl;
  fout.close();

  ifstream fin ( filename );
  if ( ! fin ) { cerr << " can't open output - " << filename <<
     endl;  return 1; }

  double pi;
  fin >> pi;
  fin.close();
  cout << " pi = " << pi << endl;

  return 0;
}
```

# Input e Output su stringhe

```
#include <sstream>
#include <iomanip>

int main()
{
  ostringstream sout;
  sout.setf(ios::fixed);
  sout << setprecision(6) << 3.1415 << endl;
  string s = sout.str();
  cout << s;

  istringstream sin( s );
  double pi;
  sin >> pi;

  cout << pi << endl;
}
```