Programmazione C++

Gianluigi Ciocca, Luigi Celona

C++11
Nuovi tipi di dati

- long long
  - Possibile definire un intero 64 bit
- nullptr
  - Tipo specifico per puntatore a null. Elimina ambiguità tra puntatore a null e intero 0.
- enum classes
  - Gli enumerativi possono essere dotati di tipo univoco (non più considerati come semplici interi)

```
enum class mele {smith=1, golden, fuji};

enum class pere {williams=1, kaiser, decana};

mele m = mele::smith;

pere p = pere::williams;

p==m; // errore
```

C++11
Keyword auto

- Keyword **auto**
  - Permette di dedurre automaticamente il tipo di una variabile
  - Solo in inizializzazione

```
auto v = 10;  // v è di tipo intero

double funct() {
 double res;
 ...
 return res;
}

auto v1 = 10;  // v1 è di tipo intero

auto v2 = funct();  // v2 è di tipo double
```

  - Particolarmente utile per gli iteratori...

```
auto i = objclass.begin();

// invece di un potenziale

std::myclass<obj, compare_obj>::iterator i = objclass.begin();
```

C++11
decltype

- Permette di ottenere il tipo di una variabile

```
double var1 = 3.14;

decltype(var1) var2; // var2 è di tipo double
```

- Utile in congiunzione con auto e per mantenere coerenza tra tipi di variabili

```
auto var1 = 3.14f;

decltype(var1) var2; // var2 è di tipo double
```

C++11
Alias di tipi con using

- **using** ha un uso simile a typedef ma è più flessibile e intuitivo
  - Al contrario di typedef può essere usato per dare dei nomi a tipi templati!

using **nome** = *tipo_di_dato*

template <typename T> using **nome** = *tipo_di_dato<T>*

```
typedef myclass<data_type, functor> mytype;

using mytype1 = myclass<data_type, functor>;

template <typename F> using mytype2 = myclass<int, F>;

mytype2<double> var;
```

C++11
static_assert

- Le assert vengono valutate a runtime
  - Servono per verificare certe condizioni
  - Es. test, pre-condizioni, post-condizioni, ...

- Le static_assert vengono valutate a compile-time
  - Servono a valutare delle condizioni NOTE in fase di compilazione
  - Es. versioning, dimensione dati, ...

```
static_assert(condizione, stringa_errore)
```

```
#include <cassert>

static_assert(sizeof(int)==8, "Dimensioni int non supportate");
```

C++11
Lambda Expressions (1)

- Funzioni Lambda
  - Sono funzioni create “al volo”
  - Simili ai funtori
    - Ma non serve creare un oggetto
  - Sono espressioni “usa e getta”
    - Anche se è possibile assegnarle ad una variabile per riuso
  - Spesso si usano in cicli o per specificare policy

C++11
Lambda Expressions (2)

- Sintassi

```
[ capture clause ] ( parameters ) -> return-type { body }
```

  - Il valore di ritorno è opzionale

```
// Espressione che valuta due interi e ritorna true se il primo è minore del secondo
[] (int a, int b) -> bool {return a<b;}

// Espressione che cattura una variabile esterna (vector) da usare per confronto
int limit;
[limit] (int a,int b) -> bool {return (a<b) && (a<limit);}
```

C++11
Lambda Expressions (3)

- Capture Clause
  - Specifica se e quali variabili nello scope esterno (enclosing scope) l'espressione sono visibili/usabili all'interno del body
  - [ ] Non catturare nessuna variabile
  - [var-name] Cattura la variabile var-name per valore/copia
  - [=] Cattura TUTTE le variabili per valore/copia
  - [&var-name] Cattura la variabile var-name per reference
  - [&] Cattura TUTTE le variabili per reference
  - [this] Cattura tutte le variabili membro di una classe

  - La cattura per reference permette di modificare il contenuto della variabile catturata

C++11
Lambda Expressions (4)

- Esempi

```
#include <iostream>
#include <algorithm>

class functor {
 double inner;
public:
 functor(double d) : inner(d) {}

 double operator()(double v) const {
          return v*inner;
 }
};

int main() {

 double array[5] = {1.0,2.0,3.0,4.0,5.0};

 std::transform(array,array+5,array,functor(3)); // array={3.0,6.0,9.0,12.0,15.0}

 return 0;
}
```

C++11
Lambda Expressions (5)

- Esempi

```
#include <iostream>
#include <algorithm>

int main() {

 double array[5] = {1.0,2.0,3.0,4.0,5.0};

 double inner = 3;

 std::transform(array,array+5,array,[=](double d){return d*inner;});
 // array={3.0,6.0,9.0,12.0,15.0}

 auto lambda = [=](double d){return d*inner;};

 std::transform(array,array+5,array,lambda); // array={9.0,18.0,27.0,36.0,45.0}

 auto lambda2 = [](double &d) {d=0;};

 std::for_each(array,array+5,lambda2); // array={0.0,0.0,0.0,0.0,0.0}

 return 0;
}
```

C++11

Iteratori: cbegin e cend

- E' possibile ora chiedere esplicitamente un const_iterator anche ad un oggetto non costante
  - cbegin(): ritorna l'iteratore costante di inizio sequenza
  - cend(): ritorna l'iteratore costante di fine sequenza

```
auto i = objclass.begin(); // iterator

auto ci = objclass.cbegin(); // const_iterator
```

C++11
Initializer list

- E' possibile inizializzare un oggetto (tipicamente container) con una lista di valori dello stesso tipo

```
std::vector<int> vi = {1,2,3,4,5,6,7};

// vi.size() == 7;
```

- Le classi container devono possedere un costruttore speciale

```
#include <initializer_list>

class classobj {

public:

 classobj(const initializer_list<int> &il) {
   // il.begin(), il.end() per accedere ai dati e inserirli nella classe
 }

};
```

C++11
Range-based for

- E' possibile eseguire i cicli for in forma più compatta

```
std::vector<int> vi = {1,2,3,4,5,6,7};

for(auto i=vi.begin(), ie = vi.end(); i!=ie; ++i) { // i è un iterator
  //...
}

for(auto i : vi) { // i è un intero (copia)
  // le modifiche a i non si propagano al container
}

for(auto &i : vi) { // i è un intero (reference)
  // le modifiche a i si propagano al container
}
```

C++11
Costruttori delega

- In un costruttore di una classe è possibile richiamare un altro costruttore della stessa classe
  - Deve essere chiamato nella initialization list

```
class myclass {
//...
public:

 myclass() {// ... };

 myclass(int,int) : myclass() {// ... };

};
```

C++11
Funzioni membro default e delete (1)

- L'obiettivo è rendere più chiaro come sono o non sono definite delle funzioni membro

```
ret_type nome_funzione(parametri) = default/delete;
```

- **delete** indica esplicitamente che un metodo non è implementato e, per i fondamentali, non deve essere neanche sintetizzato dal compilatore

- **default** indica esplicitamente che un metodo fondamentale deve essere sintetizzato dal compilatore

C++11
Funzioni membro default e delete (2)

- Esempio

```
class myclass {
 //...
public:

 // Il costruttore di default è sintetizzato dal compilatore
 myclass() = default;

 // Il costruttore di copia non esiste: impedito l'uso
 myclass(const myclass &other) = delete;

};

myclass c1;

Myclass c2(c1); // errore
```

C++11
Funzioni membro override e final (1)

- L'obiettivo è rendere più chiaro come possono essere usate le funzioni membro ridefinite, in una gerarchia di classi
  - Si applicano alle classi polimorfe

```
[virtual] ret_type nome_funzione(params) override/final;
```

- **override** indica esplicitamente che un metodo della classe derivata è la versione ridefinita di un metodo della classe padre

- **final** indica esplicitamente che un metodo non può essere ridefinito nella classe derivata

C++11
Funzioni membro override e final (2)

- Esempio

```
class padre {
public:

 virtual void funct_A() final;

 virtual void funct_B();

};

class figlia : public padre {
public:

 void funct_A() ovveride; // ERRORE

 void funct_B() override; // OK
};
```

C++11
Move semantic (1)

- Nelle classi sono definiti due ulteriori metodi con semantica «move»
  - Operano tra this e un oggetto other dello stesso tipo
  - I dati di other sono spostati in this
  - this in sostanza si appropria dei dati di other perdendo i propri
  - other potrebbe diventare un oggetto non più valido (oppure vuoto)
- La move semantic permette di ridurre le operazioni di copia tipiche dell'assegnamento e copy constructor
  - Infatti i due nuovi metodi sono:
    - Move copy constructor
    - Move assegnamento

C++11
Move semantic (2)

- Move copy constructor

```
nome_classe(nome_classe &&other);
```

  - NOTA: doppio ref & e manca const sul parametro!

```
class myclass {
 //...
public:

 myclass(myclass &&other) {
   // Logica di move
 }

};
```

C++11
Move semantic (3)

- Move assegnamento

```
nome_classe& operator=(nome_classe &&other);
```

  - NOTA: doppio ref & e manca const sul parametro!

```
class myclass {
 //...
public:

 myclass&& operator=(myclass &&other) {
   // Logica di move
   return *this;
 }

};
```

C++11
Move semantic (4)

- Quale è la logica di move da scrivere?
  - Può essere un semplice scippo dei puntatori (es. nel cctor)
  - Può essere qualcosa che ricorda l'uso della swap (es. in op=)
  - Si possono usare operazioni di move sui dati membro

```
class myclass {
 int *array;

public:

 myclass(myclass &&other) {
   array = other.array;
   other.array = nullptr; // other diventa non più valido!
 }

 myclass& operator=(myclass &&other) {
   std::swap(other.array,array);
   return *this;
 }

};
```

C++11
std::array

- Array statico (sullo stack) con interfaccia simile ai container della STL

```
#include <array>

int main() {

 std::array<int,10> = {1,2,3};

 return 0;
}
```

C++11
std::chrono (1)

- Sono definiti diverse nuove classi per la gestione del tempo nel namespace std::chrono
- **std::chrono::system_clock** tempo di sistema che può cambiare se l'utente cambia il time di sistema (sconsigliato)
- **std::chrono::steady_clock** tempo che è garantito non decrescere mai (consigliato)
- **std:chrono::high_resolution_clock** tempo ad alta risoluzione. L'implemntazione dipende dal compilatore

C++11
std::chrono (2)

- Esempi

```
#include <iostream>
#include <chrono>

int main() {

 // start è di tipo std::chrono::steady_clock::time_point
 auto start = std::chrono::steady_clock::now();

 //...

 auto end = std::chrono::steady_clock::now();

 auto duration = end-start;

 auto secs = std::chrono::duration_cast<std::chrono::seconds>(duration).count();

 std::chrono::time_t tme=std::chrono::steady_clock::to_time_t(start);

 std::cout << "inizio computazione: " << std::ctime(&tme) << std::endl;
 std::cout << "Durata: " << secs << std::endl;
 return 0;
}
```

C++11
Smart pointers (1)

- Sono introdotti tre tipi di smart pointers che contengono una logica per la gestione dei puntatori raw
  - **std::shared_ptr<T>**
    - Ownership dei dati con reference counting. I dati possono essere condivisi tra più puntatori. Memoria automaticamente deallocata quando reference counting và a zero. Possono essere copiati e assegnati.
  - **std::unique_ptr<T>**
    - Ownership del puntatore al dato. Non possono essere copiati o assegnati. Memoria automaticamente deallocata quando unique_ptr esce di scope.
  - **std::weak_ptr<T>**
    - Non hanno ownership dei dati. Usati solo per verificare esistenza/validità di un shared_ptr.

C++11
Smart pointers (2)

- Esempi

```
#include <iostream>
#include <memory>

void f1(std::unique_ptr<int> ptr) {}

void f2(std::unique_ptr<int> &ptr) {}

int main() {

  std::unique_ptr<int> uptr(new int);

  uptr = std::unique_ptr<int>(new int); // errore

  f1(uptr); // errore

  f2(uptr); //ok

  int val = *uptr;

  *uptr = 100;

  return 0;
}
```

C++11
Smart pointers (3)

- Esempi

```
#include <iostream>
#include <memory>

struct obj {
  int value;
};

void f1(std::shared_ptr<obj> ptr) {}

int main() {

  std::shared_ptr<obj> uptr(new obj);

  uptr = std::shared_ptr<obj>(new obj); // ok

  f1(uptr); // ok

  sptr->value = 100;

  return 0;
}
```

C++11
Smart pointers (4)

- Esempi

```
#include <iostream>
#include <memory>

struct obj {
 int value;
};

std::weak_ptr<obj> wptr;

int main() {

 {
   std::shared_ptr<obj> sptr(new obj);

   wptr = sptr; // ok

   wptr.expired(); // = false
 }

 wptr.expired(); // = true

 return 0;
}
```

C++14
Nuove funzionalità (1)

- **Binary literals:** rappresentazione di numeri binari
  - Iniziano con il prefisso **`0b`** o **`0B`**
  - Seguiti da una sequenza di uno o più cifre binarie 0 o 1
  - Sono di tipo *int*

```
0b110 → 6
0b11111111 → 255
0B1101 → 13
```

- Apice per separare le cifre

```
long decval = 1'048'576;                          // gruppi di tre cifre
long hexval = 0x10'0000;                          // gruppi di quattro cifre
long octval = 00'04'00'00'00;                     // gruppi di due cifre
long binval = 0b100'000000'000000'000000;         // gruppi di sei cifre
```

C++14
Nuove funzionalità (2)

- **Lambda expressions generiche**
  - È possibile specificare il tipo **`auto`** consentendo di creare delle lambda expressions *polimorfe*

```
auto identity = [](auto x) { return x; };

int three = identity(3);                    // == 3
std::string foo = identity("foo");          // == "foo"
```

- Attributo **`[[deprecated]]`**
  - Indica che una unità (funzione, classe, ecc.) è obsoleta
  - Può essere specificato un messaggio di warning

```
[[deprecated]]
void old_method();

[[deprecated("Use new_method instead")]]
void legacy_method();
```

C++14
Nuove funzionalità (3)

- Variable templates
  - Le variabili possono essere template

```
template<class T>
constexpr T pi = T(3.1415926535897932385);

template<class T>
constexpr T e = T(2.7182818284590452353);
```

- Deduzione del tipo restituito
  - In caso di utilizzo di **auto**, in funzioni lambda ad esempio, il compilatore tenta di dedurre il tipo restituito

```
auto f(int i){ // deduce che il tipo restituito è int
    return i;
}
```

C++14
Decltype(auto)

- L'identificatore di tipo **`decltype(auto)`** deduce anche un tipo come fa **`auto`**
  - A differenza di **`auto`** deduce i tipi di ritorno mantenendo i loro riferimenti

```
const int x = 0;
auto x1 = x;                          // int
decltype(auto) x2 = x;                // const int
int y = 0;
int& y1 = y;
auto y2 = y1;                         // int
decltype(auto) y3 = y1;               // int&
```

C++14
Compile-time integer sequences

- La classe template **`std::integer_sequence`** consente di creare in fase di compilazione una sequenza di interi
  - **`std::make_integer_sequence<T, N...>`** crea una sequenza di 0,…,N-1 elementi di tipo T
  - **`std::index_sequence_for<T...>`** converte un template in una lista di interi

```
template<typename Array, std::size_t... I>
decltype(auto) a2t_impl(const Array& a, std::integer_sequence<std::size_t, I...>)
{
     return std::make_tuple(a[I]...);
}


template<typename T, std::size_t N, typename Indices = std::make_index_sequence<N>>
decltype(auto) a2t(const std::array<T, N>& a) {
     return a2t_impl(a, Indices());
}
```