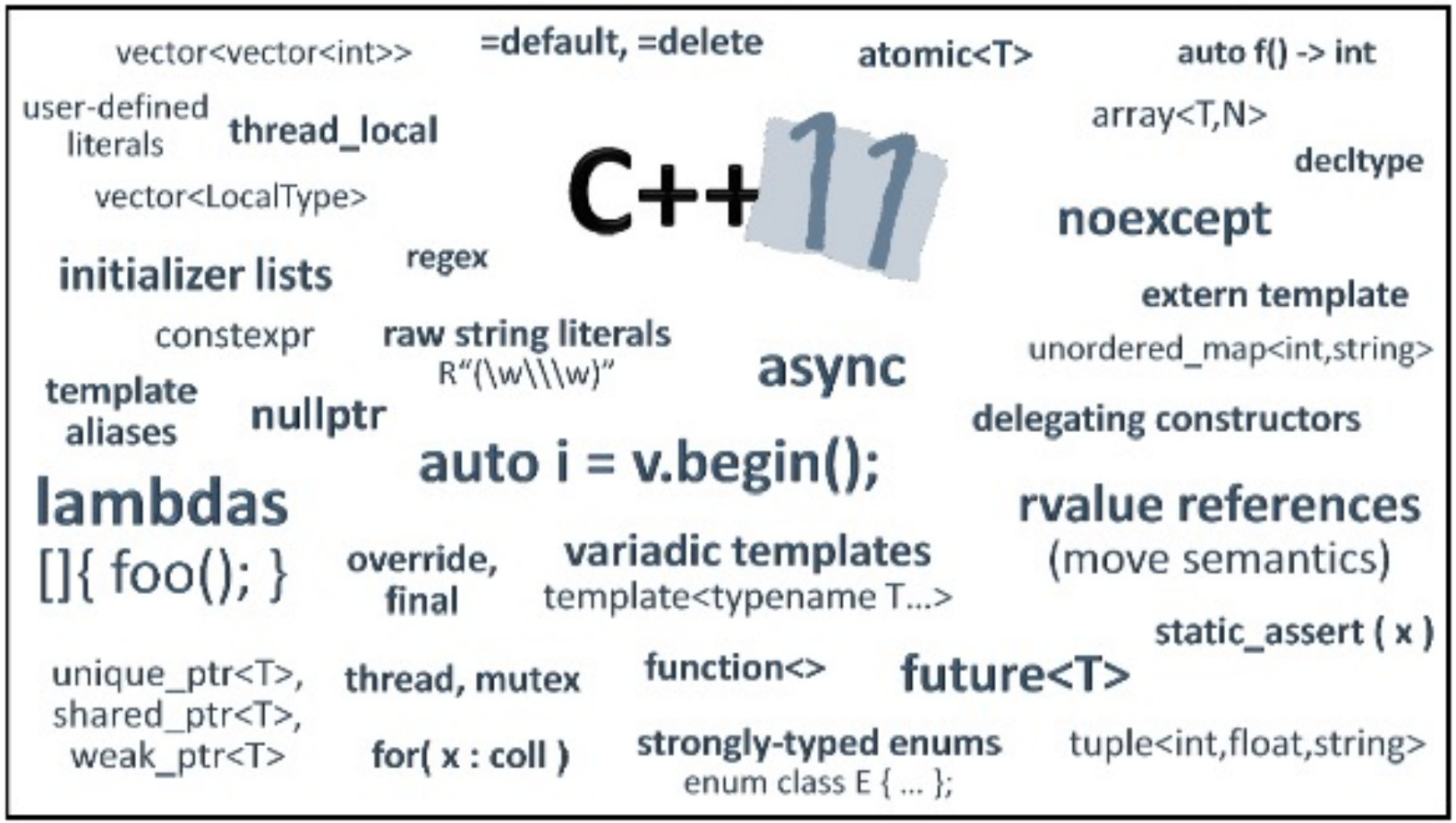

vector<vector<int>>
=default, =delete
atomic<T>
auto f() -> int
user-defined literals
thread_local
array<T,N>
C++11
decltype
vector<LocalType>
noexcept
regex
initializer lists
extern template
constexpr
raw string literals
R"(\w\\\w)"
async
unordered_map<int,string>
template aliases
nullptr
delegating constructors
auto i = v.begin();
lambdas
[]{ foo(); }
rvalue references
(move semantics)
override, final
variadic templates
template<typename T...>
static_assert (x)
unique_ptr<T>,
shared_ptr<T>,
weak_ptr<T>
thread, mutex
function<>
future<T>
for(x : coll)
strongly-typed enums
enum class E { ... };
tuple<int,float,string>

Compilazione C++11 (dalla versione 4.7)

```
g++ -std=c++11
```

Specifica delle eccezioni deprecata

Problemi nella specifica delle eccezioni

• **Run-time checking**: il test di conformità delle eccezioni avviene a run-time e non a compile-time, quindi non vi è una garanzia statica di conformità.

• **Run-time overhead**: Run-time checking richiede al compilatore del codice addizionale che può inficiare alcune ottimizzazioni.

• **Inutilizzabile con i template**: in generale i parametri di tipo dei template non permettono di specificare le eccezioni.

Inferenza automatica di tipo

Seccature dello strong typing...

```
vector< vector<int> >::const_iterator cit=v.begin();
```

Keyword: **auto**

Dichiarazioni di variabili senza specifica del loro tipo.

```
auto x = 0;   // x ha tipo int perché 0 è un litterale di tipo int
auto c = 'f'; // char
auto d = 0.7; // double
auto debito_nazionale = 2900000000000L; // long int
auto y = qt_obj.qt_fun(); // y ha il tipo di ritorno di qt_fun
```

Permette di evitare alcune verbosità dello strong typing, specialmente per i template.

```
void fun(const vector<int> &vi){
 vector<int>::const_iterator ci=vi.begin();
 ...
}

// posso rimpiazzarlo con

void fun(const vector<int> &vi){
 auto ci=vi.begin();
 ...
}
```

```
void fun(vector<int> &vi){
 auto ci=vi.begin(); // tipo: vector<int>::iterator
 ...
}
```

Keyword: **`decltype`**

Determina staticamente il tipo di espressioni.

```
int x = 3;
decltype(x) y = 4;
```

```
std::vector<int> v(1);
auto a = v[0];          // a ha tipo int
decltype(v[1]) b = 1; // b ha tipo int
auto c = 0;             // c ha tipo int
auto d = c;             // d ha tipo int
decltype(c) e;          // e ha tipo int
decltype(0) f;          // f ha tipo int
```

Inizializzazione uniforme aggregata con { }

Inizializzazione uniforme per array

```
// inizializzazione di array dinamico
int* a = new int[3] {1,2,0};

class X {
  int a[4];
public:
  X() : a{1,2,3,4} {} // inizializzazione di campo dati array
};
```

Inizializzazione uniforme per contenitori STL

```
// inizializzazione di contenitori in C++11
std::vector<string> vs = {"first", "second", "third"};

std::map<string,string> singers =
  { {"Federer", "347 0123456"},
    {"RogerWaters", "348 9876543"} };

void fun(std::list<double> l);
fun({0.34, -3.2, 5, 4.0});
```

keywords `default` e `delete`

Per ogni classe sono disponibili le versioni **standard** di:

1) costruttore di default

2) costruttore di copia

3) assegnazione

4) distruttore

In C++11 tali funzioni standard si possono rendere esplicitamente di default oppure non disponibili.

```
class A {
public:
 A(int) {}                  // costruttore ad 1 argomento
 A() = default;             // costruttore altrimenti non disponibile
 virtual ~A() = default; // distruttore virtuale standard
};
```

```
class NoCopy {
public:
 NoCopy& operator=(const NoCopy& ) = delete;
 NoCopy (const NoCopy&) = delete;
};

int main() {
  NoCopy a,b;
  NoCopy b(a); // errore in compilazione
  b=a;         // errore in compilazione
}
```

```
class OnlyDouble {
public:
 static void f(double) {}
 template <class T> static void f(T) = delete;
 // NESSUNA CONVERSIONE A DOUBLE PERMESSA
};

int main() {
 int a=5; float f=3.1;
 OnlyDouble::f(a); // ILLEGALE: use of deleted function with T=int
 OnlyDouble::f(x); // ILLEGALE: use of deleted function with T=float
}
```

Overriding esplicito

keyword: **override**

Per dichiarare esplicitamente quando si definisce un overriding di un metodo virtuale

```
class B {
public:
  virtual void m(double) {}
  virtual void f(int) {}
};

class D: public B {
public:
  virtual void m(int) override {} // ILLEGALE
  virtual void f(int) override {} // OK
};
```

Serve per evitare di definire, o di dimenticare, inavvertitamente degli overriding

keyword: **final**

Un metodo virtuale **final** proibisce alle classi derivate di effettuare overriding

```
class B {
public:
  virtual void m(int) {}
};

class C: public B {
public:
  virtual void m(int) final {} // final overrider
};

class D: public C {
public:
  virtual void m(int) {}; // ILLEGALE
};
```

"keyword" **`override final`**

Note that neither `override` nor `final` are language keywords. They are technically identifiers; **they only gain special meaning when used in those specific contexts. In any other location, they can be valid identifiers.**

`final` **può** permettere al compilatore
una ottimizzazione di de-virtualizzazione.
Esercizio: verificare su g++/clang

Puntatori nulli

```
void f(int);
void f(char*);

int main() {
  f(0);          // quale f invoca? invoca f(int)
}
```

keyword: **nullptr**
Sostituisce la macro **NULL** ed il valore 0.

nullptr ha come tipo **std::nullptr_t** che è convertibile implicitamente a **qualsiasi tipo puntatore ed a bool**, mentre non è convertibile implicitamente ai tipi primitivi integrali

```
void f(int);
void f(char*);

int main() {
  f(nullptr);    // quale f invoca? invoca f(char*)
}
```

```
const char* pc = str.c_str();
if (pc != nullptr) std::cout << pc << endl;
```

Chiamate di costruttori

Un costruttore nella sua lista di inizializzazione può invocare un altro costruttore della stessa classe, un meccanismo noto come **delegation** e disponibile in linguaggi come Java

```
class C {
 int x, y;
 char* p;
public:
 C(int v, int w) : x(v), y(w), p(new char [5]) {}
 C(): C(0,0) {}
 C(int v): C(v,0) {}
};
```

È una alternativa al meccanismo degli argomenti di default dei costruttori; alternativa considerata **preferibile** da alcuni esperti di programmazione.

Funtori

Funtori

Un funtore è un oggetto di una classe che può essere trattato come fosse una funzione (o un puntatore a funzione):

```
FunctorClass fun;
fun(1,4,5);
```

È possibile fare ciò mediante l'overloading di `operator()`, l'operatore di "chiamata di funzione": può avere un qualsiasi numero di parametri di qualsiasi tipo e ritornare qualsiasi tipo. Quando si invoca `operator()` su un oggetto, si può quindi pensare di "invocare" quel funtore.

```
class FunctorClass {
private:
  int x;
public:
 FunctorClass(int n): x(n) {}
 int operator() (int y) const {return x+y;}
};

int main() {
  FunctorClass sommaCinque(5);
  cout << sommaCinque(6); // stampa 11
}
```

```
class MoltiplicaPer {
private:
  int factor;
public:
  MoltiplicaPer(int x): factor(x) {}
  int operator() (int y) const {return factor*y;}
};

int main() {
  vector<int> v;
  v.push_back(1); v.push_back(2); v.push_back(3);
  cout << v[0] << " " << v[1] << " " << v[2]; // stampa 1 2 3
  std::transform (v.begin(), v.end(), v.begin(), MoltiplicaPer(2));
  cout << v[0] << " " << v[1] << " " << v[2]; // stampa 2 4 6
}
```

```
template <class InputIterator, class OutputIterator, class UnaryOperation>
  OutputIterator transform (InputIterator first, InputIterator last,
                            OutputIterator result, UnaryOperation op);

/*
Applica op ad ogni elemento in [first,last) e memorizza il valore ritornato da ogni
applicazione di op nel segmento di contenitore che inizia da result.
Equivalente al seguente codice:
*/

template <class InputIterator, class OutputIterator, class UnaryOperation>
  OutputIterator transform (InputIterator first, InputIterator last,
                            OutputIterator result, UnaryOperation op) {
  while (first != last) {
    *result = op(*first);
    ++result; ++first;
  }
  return result;
}
```

```
class UgualeA {
private:
  int number;
public:
  UgualeA(int n): number(n) {}
  bool operator() (int x) const {return x==number;}
};

// template di funzione, con parametro di tipo "funtore" int -> bool
template<class Functor>
vector<int> find_matching(const vector<int>& v, Functor pred) {
  vector<int> ret;
  for(vector<int>::const_iterator it = v.begin(); it<v.end(); ++it)
    if( pred(*it) ) ret.push_back(*it); // deve essere disponibile bool operator()(int)
  return ret;
}

int main() {
  vector<int> w;
  w.push_back(1); w.push_back(5); w.push_back(1); w.push_back(3);
  vector<int> r = find_matching(w, UgualeA(1));
  for(int i=0; i<r.size(); ++i) cout << r[i] << " ";
  // stampa 1 1
};
```

Altro esempio d'uso di funtori con `std::for_each`

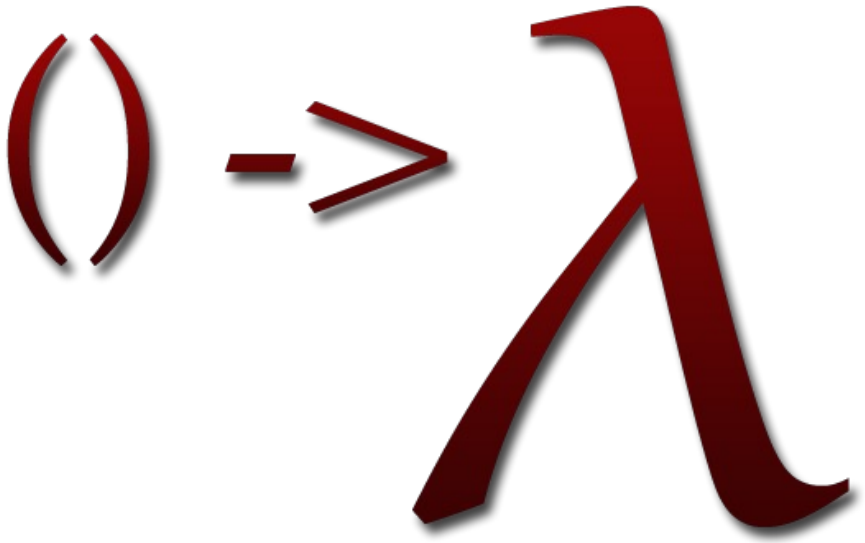
```
class Functor {
  int fattore;
public:
  Functor(int m=1): fattore(m) {}
  void operator() (int x) const {cout << fattore*x << " ";}
};

void fun1(const vector<int>& v) {
  Functor f(2); // funtore int -> void
  std::for_each(v.begin(), v.end(), f);
}
```

```
std::for_each(InputIterator first, InputIterator last, UnaryFunction f)
```

è un template di funzione di STL dichiarata in **`<algorithm>`**

Lambda espressioni o closures

Alias: funtori anonimi



Funzione anonima locale

`[capture list] (lista parametri) ->return-type { corpo }`

`(lista parametri)` e `->return-type` sono opzionali

```
[] ->int {return 3*3;}          // lista vuota di parametri

[](int x, int y) {return x+y;} // tipo di ritorno implicito: int

[](int x, int y) ->int {return x+y;} // tipo di ritorno esplicito: int

[](int& x) {++x;}               // tipo di ritorno implicito: void

[](int& x) ->void {++x;}        // tipo di ritorno esplicito: void
```

Closure

`[capture list]` elenca la lista delle variabili della closure, cioè variabili all'esterno della lambda espressione usate come l-valore (lettura e scrittura) o r-valore (sola lettura) dalla lambda espressione.

Closure (computer programming)

From Wikipedia, the free encyclopedia

In programming languages, **closures** (also **lexical closures** or **function closures**) are techniques for implementing lexically scoped name binding in languages with first-class functions. Operationally, a closure is a record storing a function[a] together with an environment:[1] a mapping associating each free variable of the function (variables that are used locally, but defined in an enclosing scope) with the value or reference to which the name was bound when the closure was created.[b] A closure—unlike a plain function—allows the function to access those *captured variables* through the closure's copies of their values or references, even when the function is invoked outside their scope.

Closure

`[capture list]` elenca la lista delle variabili della closure, cioè variabili all'esterno della lambda espressione usate come l-valore (lettura e scrittura) o r-valore (sola lettura) dalla lambda espressione.

```
[]       \\ nessuna variabile esterna catturata
[x, &y]  \\ x catturata per valore, y per riferimento
[&]      \\ tutte le variabili esterne catturate per riferimento
[=]      \\ tutte le variabili esterne catturate per valore
[&, x]   \\ tutte le variabili per riferimento, tranne x per valore
```

Esempio

```
#include<algorithm>     // dichiarazione del template for_each
for_each(InputIterator first, InputIterator last, UnaryFunction f)

// funz. che ritorna true se e solo se c è "maiuscola"
bool is_upper(char c);

int main() {
  char* s = "Hello World";
  int UppercaseNum = 0; // nella closure della lambda espressione
  std::for_each(s, s+sizeof(s), [&UppercaseNum] (char c) {
                                    if (is_upper(c)) UppercaseNum++;
                                  }
                );

 cout<< UppercaseNum << " lettere maiuscole in: " << s << endl;
}
```

`this` può essere catturato solo per valore

```
class C {
  ...
  int f() const {...}

  int m(const vector<int>& v) const {
    int totale = 0;
    int a = someClass::getSomeIntValue();
    std::for_each(v.begin(), v.end(), [&totale, a, this](int x) {
      totale += x * a * this->f();
    } );
    return totale;
  }
};
```

Esempio: capture di un parametro

```
void fun(const std::vector<int>& v, int fattore) {
  std::for_each(v.begin(), v.end(),
     [fattore](int x) {std::cout << fattore*x << " ";} )
}
```

```
void fun(std::vector<double>& v, double epsilon){
  std::transform(v.begin(), v.end(), v.begin(),
       [epsilon](double d) -> double {
         if (d < epsilon) { return 0; }
         else { return d; }
       }
  );
}
```

Esempio

```
class RubricaEmail {
private:
  vector<string> rub; // rubrica di email
public:
  // un template di metodo permette di istanziare sia a funtori che a lambdas
  // Functor con parametro const string& che ritorna un bool
  template<class Functor>
  vector<string> trovaIndirizzi(Functor test) const {
    vector<string> ris;
    for(auto it = rub.begin(); it != rub.end(); ++it)
      if (test(*it)) ris.push_back(*it);
    return ris;
  }
};

vector<string> trovaIndirizziGmail(const RubricaEmail& r) {
  return r.trovaIndirizzi(
       [] (const string& email) {
                                  return email.std::find("@gmail.com") != string::npos;
                                 }
                          );
}

vector<string> trovaIndirizziConMatch(const RubricaEmail& r, const string& match) {
  return r.trovaIndirizzi(
       [match] (const string& email) {
                                       return email.std::find(match) != string::npos;
                                      }
                          );
}
```