Note dalle lezioni

Per chi non avesse seguito le lezioni, il riassunto presentato in questa pagina potrebbe essere fin troppo sintetico e sbrigativo in alcuni punti; si consiglia pertanto di irrobustire la propria preparazione con altre risorse, come ad esempio le lezioni dei professori, oppure anche cercando articoli e spiegazioni dei singoli argomenti su siti esterni.

Qui alcuni appunti tratti dalle lezioni del corso (anno 2020/2021):

- *note lezioni*: https://samaretas.github.io/linkedNotes/programming_languages/C++/Note%20lezioni.pdf
- *note_esercitazioni*: https://samaretas.github.io/linkedNotes/programming_languages/C++/Note%20esercitazioni.pdf

Riassunto

Questo riassunto invece può essere comodo per il ripasso e per essere utilizzato come prontuario in sede d'esame, dal momento che questo è open book.

1. Base

Puntatore = variabile che contiene un indirizzo

Tre tipi di passaggio parametri:

- valore
- indirizzo (`*`)
- riferimento (`&`)

Incremento pre/post fisso:

- `++(++i)` incrementa due volte
- `(i++)++` incrementa solo una volta (la seconda viene incrementata la variabile temporanea di copia)
- `(a = 1)++` a=2: reference serve per le modifiche successive

Possiamo dichiarare una variabile in due modi:

- `int valint(34);` come classi;
- `int valint2 = 34;` come al solito.

Le `struct` hanno attributi sempre accessibili dall'esterno (usare le classi).

Definire come `const` quei metodi che non fanno modifiche all'oggetto.

Quando definiamo un vettore di oggetti, ciascuno di essi viene inizializzato con il costruttore di default.

La distruzione di oggetti nello stack avviene nell'ordine inverso della dichiarazione: `p1`, `p2`, `p3` vengono distrutti secondo l'ordine `p3`, `p2`, `p1`.

La parola chiave *inline* si applica alle definizione di funzioni o funzioni membro come forma di ottimizzazione. Essa è una speciale direttiva al compilatore che, se eseguita, consiste nel sostituire la chiamata a funzione con il corpo della funzione stessa. In molti casi, per funzioni composte da poche istruzioni semplici e invocate frequentemente, ciò può comportare un incremento delle prestazioni, a scapito di un aumento delle dimensioni dell'eseguibile. Tuttavia, ciò non è sempre vero. Rendere una funzione effettivamente inline dipende dal compilatore C++, che applicando le sue euristiche sul codice, può eseguire questa direttiva o ignorarla, così come applicarla a metodi o funzioni che non abbiamo esplicitamente definito come inline.

1.0.1. Parametri facoltativi

```
class Persona {
  Persona(const string& no, const string& co, int _eta = 0);
};

Persona::Persona(const string& no, const string& co, int _eta){ ... }
```

1.0.2. Static e attributi costanti

```
class A {
  private:
    // ATTRIBUTI COSTANTI: con alcuni compilatori è possibile
    // inizializzare qui solo se int o solo se float
    const int d; // = 99;
    const float e; // = 3.1415;

    // ATTRIBUTI STATICI
    // comune a tutte le istanze della classe A
    // NON possiamo inizializzarli qui e neanche nei ctor initializer
    // Dobbiamo inizializzarlo fuori dalla classe, una volta per sempre
    static int numIst;

    // Stesso valore per tutte le istanze della nostra classe
    static int const a = 1; // con integral funziona l'inizializzazione
    static const float b; //non si può inizializzare perché virgola mobile
  public:
    // constructor initializer. Stiamo usando costruttore di int e float
    A() : d(99), e(3.1415) {
      numIst++; // ok, nessun problema
      // d = 99;  //ERRORE: non si può cambiare il valore di una costante
    }
    ~A(){ numIst--; }
    // non si può mettere qualificatore const
    static int getStatic_numIst(){
      return numIst;
    }
};
```

```
int A::numIst = 0;      // OK, inizializzazione di attributo static
const float A::b= 0.01; // OK, inizializzazione di attributo static const

int main(int argc, char** argv) {
  // cout << A::get_numIst() << endl; // ERRORE: metodo non è statico
  // non serve un'istanza per accedere ad un attributo static
  cout << A::getStatic_numIst() << endl;
  return 0;
}
```

1.0.3. Gestione dei file

```
#include <fstream>
int main(){
  ofstream out; // ifstream for input
  out.open("file.txt", ios::app); // ios::out di default
  out << "test" << endl;
  out.close();
}
```

2. Classi

Vengono creati di default dal compilatore:

- costruttore senza parametri (nota: se si crea un costruttore con qualche parametro, questo non esisterà più);
- costruttore copia (di default fa una copia per valore di tutti i campi);
- costruttore di assegnazione;
- distruttore.

```
Myclass h1(10000); // costruttore specifico
Myclass h2 = h1;   // costruttore copia
Myclass h3(h1);    // costruttore copia (modo alternativo)

Myclass h4(60000); // costruttore specifico
h1 = h4;           // operatore = assegnazione

Myclass h5 = createMyclass(1000); // costrut copia + distrut(1000)
h5 = createMyclass(500); // operator= assegnazione + distrut(500)
                         // oppure operator= spostamento per RVO
// Return Value Optimization (RVO). I compilatori moderni sono
// in grado di rilevare che stai restituendo un oggetto in base
// al valore e applicano una sorta di scorciatoia di ritorno per
// evitare copie inutili. NOTA: RVO riguarda solo i valori di
// ritorno (output), non i parametri di funzione (input).

Myclass h6( move(h1) );    // costruttore spostamento
```

`this` = variabile che contiene puntatore all'oggetto corrente

Quando bisogna allocare dinamicamente un membro di una classe:

- allocazione memoria nel costruttore;
- deallocazione memoria nel distruttore;
- effettua copia profonda sia nell'operatore di assegnazione che nel costruttore di copia (altrimenti viene copiato il puntatore e entrambi gli oggetti fanno riferimento alla stessa posizione di memoria nell'heap, causando grossi problemi). Bisogna dunque allocare una nuova porzione di memoria e copiare il contenuto della prima nella seconda.

Per chiamare il costruttore a zero parametri non bisogna inserire le parentesi in fondo.
Gli unici parametri ad essere facoltativi sono quelli in fondo.

2.0.1. Convertitori impliciti di tipo

I costruttori con un solo parametro vengono considerati come *convertitori impliciti di tipo*.

```
class A {
  int i;
  public: A(int _i);
};
void f(A);

f(3);
A a = 4;
f(A(2)); // nulla di strano
```

Se non si vuole questo comportamento bisogna usare `explicit` sul costruttore (genera errore in fase di compilazione).

```
public: explicit A(int _i);
```

2.0.2. Riferimenti incrociati con linker

Ipotizziamo di avere una classe film che ha una lista di spettatori e uno spettatore che ha una lista di film. Come possiamo compilare un sorgente di questo genere?

```
// ============== film.h ==============
class Spettatore;
class Film {
  private: ...
  public:  ...
};

// ============ spettatore.h ============
#include "film.h"
class Spettatore{
  private: ...
  public:  ...
};
```

2.0.3. Default / delete

Possiamo specificare tramite `default` e `delete` se adottare la versione di default per un metodo o se eliminarla.

```
A (const A& _a) = default;
A& operator=(const A& _a) = delete;
```

3. Overload degli operatori

https://en.wikipedia.org/wiki/Operators_in_C_and_C%2B%2B
Non è possibile aumentare la lista degli operatori.
Possiamo definire gli operatori in uno dei due modi (*NON* entrambi contemporaneamente):

- `a = A::operator=(operator+(a1, a2))` come funzione esterna;
- `a = A::operator=(a1.operator+(a2))` come metodo.

NOTA: `A& A::operator=(const A&)` si può ridefinire solo come metodo.

Esempi:

```
class A {
  int i;
  public:
    A (int _i) { i = _i; }
    A& operator=(const A& a); // a=(b=c)
    A& operator=(const int _k); // quanti ne vogliamo
    A operator+(const A& a) const;
    A operator-();
    A& operator+=(const A& a);
    A& operator++();   // prefisso
    A operator++(int); // postfisso
    bool operator<(const A& a) const;
    A operator()(double _d) const; // ogg. funzione A a; cout << a(3.0);
    A& operator[](const int index) const; // serve la reference
    friend A operator-(const A& x, const A& y);
    friend ostream& operator<<(ostream& os, const A& a);
};
A& operator=(const A& a){
  if(this == &other)
      return *this; // evitiamo autoassegnazione
  k = a.k;
  return *this;
}

A A::operator-(){
    return A(-i);
}

A A::operator+(const A& a) const {
  // a1 = a2 + a3 OK
  // a1 = a2 + 3  OK (conversione implicita di tipo)
  // a1 = 3 + a2  NON COMPILA
```

```
  return A(i + a._i);
}

A& A::operator+=(const A& a){
  i += a.i;
  return *this;
}

A& A::operator++(){ // prefisso
  ++i;
  return *this;
}

A A::operator++(int){ // postfisso
  A temp(*this);
  ++i;
  return temp;
}

bool A::operator<(const A& a) const {
  return i < a.i;
}

A A::operator()(double _d) const { // oggetti funzione
  return A(i * _d);
}

// se non ci fosse la reference potremmo accedere solamente in lettura
// in questo modo possiamo anche modificare gli elementi
A& operator[](const int index) const {
  if (index < 0){ return m[0]; }
  if (index ⩾ dim) { return m[dim-1]; }
  return m[index];
}

A operator-(const A& x, const A& y){
  return A(x.i - y.i);
}

ostream& operator<<(ostream& os, const A& a){
  return os << "[" << a.i << "]"; // no endl
}
```

4. Ereditarietà

Ci sono tre tipi di ereditarietà:

- *public*: makes `public` members of the base class `public` in the derived class, and the `protected` members of the base class remain `protected` in the derived class;
- *protected*: makes the `public` and `protected` members of the base class `protected` in the derived class;
- *private*: makes the `public` and `protected` members of the base class `private` in the derived class.

Note: `private` members of the base class are inaccessible to the derived class.

```
class Base { };
class Derived : public Base { };
```

Sia A la classe base e B la classe derivata:

```
A a; B b;
a = b; // OK
// b = a; ERRORE
```

Nel costruttore della classe figlia:

- se non chiamo il costruttore della classe padre, allora chiama il costruttore a zero parametri del padre. Nel caso non dovesse esistere allora errore in fase di compilazione;
- bisogna chiamare un costruttore della classe padre nel ctor initializer: `Studente::Studente(const string& no, const string& co, int ma, int _eta) : Persona(no, co, _eta), matricola (ma)`

Per richiamare i metodi:

- `stampa()` se è un metodo della classe figlia;
- `Persona::stampa()` se è un metodo della classe padre. Nota: non è possibile farlo con gli operatori perché non hanno lo scope.

Dato che gli operatori sono dei veri e proprio metodi, li ritroveremo anche nelle classi figlie.

4.0.1. Keyword virtual

Abbiamo due tipi di binding:

- *early binding*: risolto dal compilatore;
- *late binding*: risolto a runtime (solo se si accede tramite puntatore o reference).

Per forzare il late binding bisogna usare puntatori e anteporre *virtual* davanti al metodo nella classe padre (da fare anche col distruttore in caso di memoria allocata dinamicamente). Non serve per il costruttore.

Se si vuole usare il late binding (classe virtuale `Personaggio` con classi figlie `Cavaliere`, `Confessore`, ecc.) bisogna necessariamente usare puntatori alla classe base (`Personaggio *pp1`).

Nel caso volessimo fare l'override dell'operatore `<<`, allora bisogna creare una funzione di appoggio virtuale di stampa. Dall'operatore andremo a chiamare quella. Altrimenti funzionerebbe solo per `Studente *a = new Studente()` e non `Persona *b = new Studente()`. A questo punto, se richiamiamo con `delete p` allora prima verrà chiamato il distruttore di Studente e poi di Persona.
Ricordiamo però che bisogna definire l'operatore sia per Persona che per Studente, altrimenti se qualcuno vuole usare direttamente uno studente allora non può.

```
// def: virtual ostream& stampaoperator(ostream& os) const;
// nota: non serve "virtual" nell'implementazione
ostream& Persona::stampaoperator(ostream& os) const {
  return os << nome << cognome << eta;
}
```

```
ostream& operator<<(ostream& os, const Persona& p) {
  return p.stampaoperator(os);
}

ostream& Studente::stampaoperator(ostream& os) const {
  return os << cognome << nome << eta << matricola;
}
ostream& operator<<(ostream& os, const Studente& s) {
  return s.stampaoperator(os);
}
```

4.0.2. Classi puramente virtuali

Una classe puramente virtuale ha almeno un metodo puramente virtuale. Una classe puramente virtuale non ha istanze (può essere invece usata come classe base). La classe figlia deve necessariamente avere il metodo definito.

```
class A {
  virtual int metodo() = 0;
};
```

Non è possibile dichiarare come virtual (altrimenti bisogna usare dei metodi di appoggio):

- costruttori normali;
- costruttori di copia;
- operatori.

4.0.3. Ereditarietà multipla

Esempio basilare

```
class Studente {
  private:
    int datoStudente;
  public:
    Studente();
    ~Studente();
    void stampa() const;
};

class Lavoratore {
  private:
    int datoLavoratore;
  public:
    Lavoratore();
    ~Lavoratore();
    void stampa() const;
};

// con questo ordine vengono chiamati i costruttori di default
// per la chiamata dei distruttori avviene il contrario
class StudenteLavoratore: public Studente, public Lavoratore {
  public:
```

```
    // prima vengono chiamati i costruttori da cui eredito, nell'ordine
    // d'ereditarietà, poi viene chiamato questo costruttore
    StudenteLavoratore(){
      cout << "creato StudenteLavoratore" << endl;
    }
    ~StudenteLavoratore(){
      cout << "distrutto StudenteLavoratore" << endl;
    }
};

int main(int argc, char** argv) {
  StudenteLavoratore sl;
  // sl.stampa(); // Errore in compilazione perché ambiguo
  sl.Studente::stampa(); // utilizza il metodo di studente
  sl.Lavoratore::stampa();  // utilizza il metodo di lavoratore
  return 0;
}
```

Diamond problem: si presenta quando due superclassi ereditano dalla stessa classe.

```
// Vengono create due copie di persona. Possiamo usare virtual per
// l'ereditarietà. A questo punto però serve avere un costruttore
// di persona a zero parametri
class Persona {
  protected:
    int eta;
  public:
    Persona(int x){ eta = x; }
    Persona(){ eta = 0; }
};

class Studente : virtual public Persona {
  public:
    Studente(int x) : Persona(x){ }
};

class Faculty : virtual public Persona {
  public:
    Faculty(int x) : Persona(x){ }
};

class TA : public Faculty, public Studente {
  public:
    // l'ordine è quello definito nella prima riga della classe
    // in questo modo il costruttore Persona viene chiamato 1 volta sola
    // in Persona viene chiamato il costruttore di default
    TA(int x) : Studente(x), Faculty(x){ }

    // in Persona viene chiamato il costruttore con 1 parametro
    // TA(int x) : Studente(x), Faculty(x), Persona(x){ }
};

int main(int argc, char** argv) {
  // ordine di chiamata dei costruttori:
```

```
  // 1) Persona (zero parametri)
  // 2) Faculty
  // 3) Studente
  // 4) TA
  TA ta(27);
  return 0;
}
```

5. Programmazione generica

In C++ si usano i *templates*, che permettono di fissare il tipo parametrico a tempo di compilazione. Permettono di scrivere codice a prescindere dal tipo di dato che verrà utilizzato durante l'esecuzione.

A lezione è stato detto che abbiamo due tipi di template. In realtà **sono la stessa identica cosa** (https://www.cplusplus.com/doc/oldtutorial/templates/):

- funzioni → `template <typename T>`;
- classi → `template <class T>`, da implementare tutto nel `.h` (dentro o fuori la classe).

Si noti che è possibile specializzare un template. Nella STL avviene nel caso di un vector di bool (occupazione di 1 bit) e metaprogramming.

```
// f ed s devono essere dello stesso tipo
template <typename T>
void my_swap(T& f, T& s) {
  T tmp = f;
  f = s;
  s = tmp;
}

int main(){
  int a=3, b=4;
  my_swap<int> (a, b);
  string s1="hello", s2="world";
  my_swap<string> (s1, s2);
  return 0;
}
```

```
template <typename T>
T min(T a, T b) {
  return a < b ? a : b;
}

// anche con diversi tipi
template <typename T1, typename T2>
T1 min(T1 a, T2 b) {
  return a < b ? a : b;
}

// IMPORTANTE: in questo caso viene preso il tipo del primo argomento
// succede anche rimuovendo il primo template
```

```
int main(){
  // deduzione dei tipi
  cout << max(2, 3);   // 3
  cout << max(3, 4.3); // 4
  cout << max(4.3, 3); // 4.3
  cout << max(int(2), double(3.14)); // 3
  cout << sizeof(double);
  // possiamo farlo anche con classi, basta che ci sia operator<
}
```

```
template <typename F, typename S>
class Pair {
  public:
    Pair(const F& f, const S& s);
    F get_first() const;
    S get_second() const;
  private:
    F first;
    S second;
};

template <typename F, typename S>
Pair<F,S>::Pair(const F& f, const S& s){
  first = f; second = s;
}

template <typename F, typename S>
F Pair<F,S>::get_first() const { return first; }

template <typename F, typename S>
S Pair<F,S>::get_second() const { return second; }
```

Implementazione di una classe con operatore di stampa:

```
template <class T> class Myarraytemp;
template<typename T>
ostream& operator <<(ostream& os, const Myarraytemp<T>&a);

template <class T>
class Myarraytemp {
  private:
    int dim;
    T* m;
  public:
    Myarraytemp() { dim = 0; m = NULL; }
    Myarraytemp(int d);
    ~Myarraytemp() { if (m =/= NULL) { delete [] m; } }

    friend ostream& operator << <>(ostream& os, const Myarraytemp<T>&a);
};

// potevamo implementare anche dentro la classe
// (conviene perché è la soluzione più semplice)
```

```
template <class T> Myarraytemp<T>::Myarraytemp(int d){
    dim = d;
    // dobbiamo implementare il costruttore zero parametri,
    // perché viene usato qui
    m = new T[dim];
}

template<typename T>
ostream& operator <<(ostream& os, const Myarraytemp<T>&a){
    for (int i=0; i<a.dim; ++i){
        os << "[" << a.m[i] << "]";
    }
    return os;
}
```

6. Move semantics

http://www.cplusplus.com/reference/utility/move/
Definizione:

- *lvalue*: valori che possono stare a sinistra delle chiamate dell'operatore di assegnazione (ha un indirizzo in memoria);
- *rvalue*: valori che possono stare a destra delle chiamate dell'operatore di assegnazione. Sono generalmente quei valori per cui non è possibile ottenere il loro indirizzo in memoria, in quanto o sono literals o sono temporanei;

Conversioni:

- da lvalue a rvalue: OK;
- da rvalue a lvalue: impossibile per progettazione.

Alcuni esempi:

```
// conversione tramite operatori
int* y = &x; // x è lvalue, &x è rvalue
// il + vuole due r value
int z = x + y; // x e y convertiti implicitamente da lvalue in rvalue

int setValue() { return 6; } // restituisce un rvalue

int global = 100; int& setGlobal(){ return global; }
setGlobal() = 400; // OK, perché restituisce un lvalue

int y = 10; int& yref = y; yref++; // y vale 11

void miaFunz(int& x) { ... }
// miaFunz(99); ERRORE
int y = 3; miaFunz(y); // OK

void miaFunz2(const int& x) { /* … */ }
miaFunz2(99); // OK, stiamo associando const lvalue a rvalue

const int& c = 3; //c = 2; ERRORE non può essere modificato!

```

```
int a = 56;
// int&& d = a; ERRORE perché a lvalue
int c = 2;
int&& d = a + c; // OK perché somma genera rvalue
d = d + 1; // OK, è modificabile perché è un lvalue, altrimenti errore
cout << "d: " << d << endl;

const int& e = a + c; // OK e = e +1; // ERRORE assignment of read-only ref
```

6.0.1. lvalue references

```
int& x = y
```

6.0.2. rvalue references

Prima del C++11

```
// int& x = 6; ERRORE perché tentiamo di convertire rvalue in lvalue
const int& x=6; // compila
```

Dopo il C++11: un rvalue reference può essere associato solamente ad un rvalue. Un rvalue reference è tecnicamente un'entità dereferenziabile, ed in quanto tale può essere modificata, può cioè comparire a sinistra dell'operatore di assegnamento, e pertanto è un lvalue

```
int&& i = 7;
i = 8;
i = l.size();
std::cout << i;
```

6.0.3. std::move (c++11)

Se non la trova prova ad importare `utility`.

Funzione template che restituisce una rvalue reference del suo argomento: `move` non sposta nulla!! Ma forza il compilatore ad interpretare il suo argomento come un rvalue reference.

Il comportamento standard indica che l'oggetto da cui sto muovendo informazioni sia lasciato in uno stato valido, ma non specificato: o lo si distrugge o gli si assegna un nuovo valore.
Serve per:

- trasferire proprietà di oggetti (es. unique_ptr);
- effettuare meno copie possibili (aumentando l'efficienza), in quanto i dati in memoria vengono trasferiti dal vecchio oggetto a quello nuovo.

```
int&& i = 7;
int j = 13;
i = std::move(j++);
i++;
cout << i << " " << j; // 14 14
int k = j++; // c'è un'operazione di copy in più
```

6.0.4. move con classi

NOTA: Sebbene un oggetto possa essere inizializzato per spostamento, mediante una chiamata all'apposito costruttore, ciò non si propaga automaticamente anche ai suoi dati membro.

Dobbiamo ridefinire:

- distruttore
- costruttore di copia
- operator=
- costruttore move: sposto le proprietà da un oggetto all'altro (responsabilità, dati, ecc.)
- move operator=

```
int main(){
  A a, a1(1);
  a = a1; // copia
  a = std::move(a1); // move constructor
}

class A {
  int i;
  B* pb;
  public:
    A();
    A(int _i, string _s);
    A(const A& _a);
    A& operator=(const A& _a); //a = (b = c)

    // parametro non deve essere const perché deve essere modificabile
    A(A&& _a);
    A& operator=(A&& _a);

    ~A();
    int get_i();
    string get_s();
    void set_s(string _s);
};

// NOTA: in qualche costruttore non viene settato i
A::A(){ i = 0; pb = NULL; }
A::A(int _i, string _s){ i = _i; pb = new B(_s); }

A::A(const A& _a){ // costruttore di copia  -> l'oggetto in partenza è vuoto
  i = _a.i;
  if (_a.pb != NULL){
    pb = new B(*(_a.pb));
  } else {
    pb = NULL;
  }
}

A& A::operator=(const A& _a){
  if (this->pb == NULL) { //oggetto chiamante non ha B
    if (_a.pb != NULL) {
      pb = new B(*(_a.pb));
```

```
    }
  } else { // l'oggetto chiamante ha B
    if (_a.pb ≠ NULL) {
      (*pb) = *(_a.pb);
    } else {
      delete pb;
      pb = NULL;
    }
  }
  return *this;
}

A::A(A&& _a){ // costruttore move → l'oggetto in partenza è vuoto
  pb = _a.pb;   // si impossessa il puntatore dell'altro
  _a.pb = NULL; // lasciamo _a in uno stato valido (evitiamo che venga
                // cancellata la memoria con una potenziale delete)
  i = _a.i;
  _a.i = 0; // consuma il parametro passato
}

A& A::operator=(A&& _a){ // move operator=
  // IMPORTANTE: controllo sull'autoassegnazione,
  // altrimenti lo si andrebbe a distruggere
  if (this == &_a){ return *this; }
  if (pb ≠ NULL){ delete pb; }
  pb = _a.pb;
  _a.pb = NULL; // lasciamo _a in uno stato valido
  _a.i = 0; // consuma il parametro passato
  return *this;
}

A::~A(){ if (pb ≠ NULL){ delete pb; }}

int A::get_i(){ return i; }
string A::get_s(){
    if (pb ≠ NULL){ return pb→get_s(); }
    else { return "pb==NULL true"; }
}
void A::set_s(string _s){
    if (pb == NULL){ pb = new B(_s); }
    else { (*pb) = B(_s); }
}
```

7. Smart pointers ()

http://www.cplusplus.com/reference/memory/
Sono delle classi template che permettono di gestire in maniera intelligente dei puntatori.
Ne abbiamo visto tre tipi:

- *unique_ptr*: in ogni istante ci può essere solo un puntatore all'oggetto (garantisce l'ownership). Il puntatore non può essere copiato, supporta solo move;
- *shared_ptr*: più puntatori che condividono ownership di una risorsa: viene deallocata quando l'ultimo viene deallocato (responsabilità è dell'ultimo). Ownership di uno shared

viene persa se viene fatta una release o se viene distrutto;
- *weak_ptr*: condivide una risorsa senza mai averne l'ownership (non aumenta `use_count()`). Per accedere al contenuto è necessario (non si può dereferenziare con `*` e `->`) copiarlo un uno shared_ptr usando un metodo `lock()`.

Vengono implementati perché sono più efficienti di una qualsiasi iterazione di garbage collector.

7.0.1. Unique_ptr

Fornisce `*` e `->`

```
#include <memory>
std::unique_ptr<int> p, p1;
p.reset(new int(54));
p1.reset(new int(23));
std::cout << *p1 << p;
(*p1) = (*p2);
p1 = std::move(p); // supportano solo la move
std::cout << *p1; // e non p (stato non definito)
p1.reset(NULL);

p.get(); // puntatore statico, ma senza ownership di deallocare
p.release(); // puntatore statico e ownership
p.swap(p1); // scambia i due puntatori
```

7.0.2. Shared_ptr

Fornisce `*` e `->`

```
#include <memory>

shared_ptr<int> p;
p.reset(new int(5));
cout << p.use_count(); // ottieni numero di owner
```

Gli `shared_ptr` condividono l'ownership solo se i nuovi puntatori vengono copiati fra di loro. Nel caso si creassero due shared a partire dallo stesso oggetto, potrebbero esserci dei gravi problemi nel caso uno dei due venga distrutto.

7.0.3. Weak_ptr

```
weak_ptr<int> wp;
shared_ptr<int> s1; s1.reset(new int (5));
wp = s1;

// unico modo per accedere a wp è passare da uno shared
shared_ptr<int> s2 = wp.lock();
```

8. Espressioni Lambda

Un'espressione lambda fornisce un modo conciso per creare oggetti funzione semplici. Un'espressione lambda è un prvalore il cui oggetto risultato è chiamato oggetto chiusura, che si comporta come un oggetto funzione. Il nome deriva dal lambda calcolo, sviluppato negli anni 30' dal matematico Alonzo Church per indagare sulle questioni relative alla logica e alla computabilità.

Note:

- Possiamo considerarle delle funzioni senza nome
- Esistono due tipi di cattura: & (per riferimento) o = (per valore/copia). Di default è per copia. Possiamo esplicitarli davanti ad ogni parametro o anteporli una volta per tutte alla lista degli elementi catturati (es. `[&, a, b]`)
- Possiamo modificare i parametri passati per valore all'interno della lambda (a meno di const): i cambiamenti non verranno riscontrati anche fuori.

La cattura di variabili tra `[]`:

- possiamo effettuare una cattura per valore (default) o per riferimento (`&`). Tramite riferimento i cambiamenti della variabile vengono portati anche fuori dalla lambda;
- la parola chiave `mutable` (opzionale) permette di modificare le variabili catturate tramite valore solo all'interno del blocco di codice della lambda expression (al di fuori non vengono portati cambiamenti). Di default (senza mutable) abbiamo un errore se tentiamo di modificare il valore di una variabile catturata.

Struttura di un'espressione lambda completa (può essere tutto vuoto):

```
[captures] (params) mutable → ret { body }
```

Esempi:

- `[captures] (params) { body }`: il tipo di ritorno viene dedotto automaticamente;
- `[capture] {body}`: nessun argomento (possono anche essere presenti ma vuote).

Nota: se viene catturato per copia allora verrà usato il valore della variabile al momento della cattura, altrimenti, nel caso di cattura per riferimento, verrà considerato il valore della variabile al momento della chiamata della funzione lambda.

```
#include <functional>
// tutta la cattura sarà per riferimento
auto x1 = [&, j](int i) → int { return i*j; };
auto x2 = [&, j](int i){ return i*j; };
cout << x1(3) << x2(4);

// function<tipo_ritorno(tipo_parametri)>
function<float()> f = [i, &j](){ return i/(1.0*j); };
cout << f();

// trova il primo valore inferiore alla soglia
auto it = find_if( vec.begin(), vec.end(),
  [threshold](int value) { return value < threshold; } );

// trasforma il contenuto di vec moltiplicandolo per num
// sovrascrive direttamente sullo stesso vettore
transform(vec.begin(), vec.end(), vec.begin(),
  [num] (const int& e) { return e * num; } );
```

```
19
20  // rimuove tutti gli elementi minori di soglia
21  // perché vengono posizionati tutti alla fine
22  vec.erase( remove_if(vec.begin(), vec.end(),
23    [soglia](int n) { return n < soglia; } ), vec.end() );
```

8.0.1. Puntatori a funzione

```
1  typedef double (*P)(double); // tipo prt a funzione di nome P
2  int (*pfunz)(double, int); // dichiarazione
3  int f1(double, int);
4  pfunz = f1;
5  cout << pfunz(2.0, 3)
```

9. Standard Template Library

9.0.1. Containers

https://www.cplusplus.com/reference/stl/
Sono classi che contengono elementi di tipo parametrico

- list (forward_list)
- vector
- set (unordered_set)
- map (unordered_map)
- queue, stack, dequeue, array

9.0.1.1. List

Alcune caratteristiche:

- Non-contiguous memory
- No pre-allocated memory. The memory overhead for the list itself is constant.
- Each element requires extra space for the node which holds the element, including pointers to the next and previous elements in the list.
- Never has to re-allocate memory for the whole list just because you add an element.
- Insertions and erasures are cheap no matter where in the list they occur.
- It's cheap to combine lists with splicing.
- You cannot randomly access elements, so getting at a particular element in the list can be expensive.
- Iterators remain valid even when you add or remove elements from the list.
- If you need an array of the elements, you'll have to create a new one and add them all to it, since there is no underlying array.

```
1  list<int> l;
2  list<int> l2(5); // dimensione 5
3  list<bool> t(3, true); // 3 elementi, tutti a true
4  list<int> l3(l); // costruttore copia
5  l = l2; // operatore di assegnazione (sono due liste separate)
6
7  l.push_back(2); l.push_front(3);
```

```cpp
for (auto& el : l){ cout << el;  el++; } // per riferimento, posso modificare

list<int>::iterator iter;
for (iter=l.begin(); iter != l.end(); ++iter){ cout << *iter; }
list<int>:: reverse_iterator riter;
for (riter=l.rbegin(); riter != l.rend(); ++riter){ cout << *riter; }

cout << l.front() << l.back();
l.pop_back(); l.pop_front();

l.insert(l.begin(), 7);
l1.insert(l1.begin(), l2.begin(), l2.end()); // insert l2 at begin of l1

l.remove(3); // elimina tutte le occorrenze
l.remove_if(item -> bool);
l.erase(l.begin());

// ricerca il primo elemento 3, eliminandolo
iter = find(l.begin(), l.end(), 3);
if (iter != l.end()) { l.erase(iter); }

// dividiamo la lista in due: dal puntatore intermedio alla fine
// nella lista vecchia e dall'inizio al puntatore intermedio
// viene chiamato sulla nuova lista creata vuota non stiamo spostando
// gli elementi in memoria, sotto viene fatto un gioco di indirizzi
list<int> l2;
l2.splice(l2.cbegin(), l, a, l.cend());

cout << l.size();  // valore intero
cout << l.empty(); // valore bool

l.sort();
l.reverse();
l2.swap(l); // vengono invertiti solo gli indirizzi (efficiente)
l.unique(); // cancella tutto tranne il primo elemento di sequenze uguali
l.merge(l2); // le liste devono già essere ordinate; l2 distrutta
l.resize(4); // rimpicciolisce il contenitore
l.clear(); // svuota la lista

// permette di costruire sul posto (evitando copie inutili)
// un elemento da un insieme di parametri (in questo caso
// funziona come una push_front). Restituisce un iteratore
// che punta alla posizione del nuovo elemento
a = l.emplace(l.begin(), 11);

// riscrive la lista, assegnando i valori contenuti in un certo range
l2.assign(7, 100); // 7 interi di valore 100
l2.assign (first.begin(),first.end()); // copia di first

// misura massima possibile che nell'attuale macchina questo
// contenitore può raggiungere in linea teorica
cout << l.max_size();

void foo(int i) { cout << i << "_" ; }
```

```
for_each(mia_lista.begin(), mia_lista.end(), &foo); // <algorithm>

// NON FATTE
emplace_front() // come emplace ma per la testa
emplace_back()  // come emplace ma per la coda
crbegin()
crend()
```

9.0.1.2. Vector

Alcune caratteristiche:

- Contiguous memory.
- Pre-allocates space for future elements, so extra space required beyond what's necessary for the elements themselves.
- Each element only requires the space for the element type itself (no extra pointers).
- Can re-allocate memory for the entire vector any time that you add an element.
- Insertions at the end are constant, amortized time, but insertions elsewhere are a costly O(n).
- Erasures at the end of the vector are constant time, but for the rest it's O(n).
- You can randomly access its elements.
- Iterators are invalidated if you add or remove elements to or from the vector.
- You can easily get at the underlying array if you need an array of the elements.

Le funzioni sono praticamente uguali alla list.

Esiste una versione specializzata per i bool per occupare meno memoria: http://www.cplusplus.com/reference/vector/vector-bool/

9.0.1.3. Set

Caratteristiche:

- Gli elementi devono aver definito `operator<`
- Searching (logarithmic in size).
- Insert and delete (logarithmic in general)
- Elements are ordered (non esiste il concetto di front e back in quanto vengono memorizzati come alberi binari di ricerca bilanciati).
- Elements are always sorted from lower to higher.
- Elements are unique.
- Possiamo inserire o rimuovere un elemento all'interno del set, ma non modificarne il suo valore.

I contenitori `set` sono generalmente più lenti dei contenitori `unordered_set` per accedere ai singoli elementi tramite la loro chiave, ma consentono l'iterazione diretta su sottoinsiemi in base al loro ordine.

Possiamo scorrerli utilizzando tutti e 4 i tipi di iteratori (`const` o meno, `reverse` o meno) e `auto` .

```cpp
#include <set>
class A {};

set<A> s; // deve essere definito operator<
s.insert(A());
bool non_presente = s.find(el) == s.end();
cout << s.length();

siiter=si.find(16);
if (siiter =/= si.end()) { si.erase(siiter); }
```

9.0.1.4. Unordered Set

I set non ordinati sono contenitori che memorizzano elementi univoci senza un ordine particolare e che consentono il recupero rapido dei singoli elementi in base al loro valore.
In un unordered_set, il valore di un elemento è allo stesso tempo la sua chiave, che lo identifica in modo univoco. Le chiavi sono immutabili, quindi, gli elementi in un unordered_set non possono essere modificati una volta nel contenitore, ma possono essere inseriti e rimossi.
Internamente, gli elementi in unordered_set non sono ordinati in un ordine particolare, ma organizzati in bucket a seconda dei loro valori hash per consentire un accesso rapido ai singoli elementi direttamente dai loro valori (con una complessità temporale media costante in media).

In generale:

- più veloci dei contenitori set per accedere ai singoli elementi tramite la loro chiave
- sebbene siano generalmente meno efficienti per l'iterazione dell'intervallo attraverso un sottoinsieme dei loro elementi.

```cpp
#include <unordered_set>
unordered_set<int> si;
unordered_set<int>::iterator siiter;
si.insert(16); si.insert(2); si.insert(1); si.insert(22);
for (const auto& item : si) { }
```

9.0.1.5. Multiset

Sono identici ai set, tranne che i multiset sono contenitori che archiviano elementi seguendo un ordine specifico e in cui più elementi possono avere valori equivalenti.

```cpp
#include <set>
multiset<int> num;
num.clear();
num.insert(4); num.insert(4);

auto it = num.find(4);
if (it =/= num.end()){ num.erase(it); }

multiset<int>::iterator it;
multiset<int>::reverse_iterator rit;
for (it = num.begin(); it =/= num.end(); ++it){ ... }
for (rit = num.rbegin(); rit =/= num.rend(); ++rit){ ... }
```

9.0.1.6. Unordered multiset

Sono dei unordered set che possono contenere più elementi uguali. Gli elementi uguali vengono raggruppati in contenitori comuni.

```
#include <unordered_set>
```

9.0.1.7. Map

Caratteristiche:

- Sono ordinati (non hanno il concetto di front e back) . Vengono salvati in memoria come alberi binari di ricerca bilanciati
- Gli elementi vengono memorizzati ordinati in memoria, perciò non possiamo parlare di posizioni.
- Ciascun elemento è un `pair<chiave, valore>`, dove il tipo della chiave deve essere ordinabile (`operator<`)

I contenitori map sono generalmente più lenti dei contenitori `unordered_map` per accedere ai singoli elementi tramite la loro chiave, ma consentono l'iterazione diretta su sottoinsiemi in base al loro ordine.

Possiamo scorrerli utilizzando tutti e 4 i tipi di iteratori (`const` o meno, `reverse` o meno) e `auto`.

```
class A {};

map<A, int> m; // deve essere definito operator<
m.insert(pair<A, int>(A(), 2));
m[A(1)] = 3; // posso modificare o inserire

map<int, A> ma; // < è definito per gli int, dunque OK
ma.insert(pair<int, A>(3, A()));
ma[3] = A(); // per le map è definito operator[]
for (auto it = m.begin(); it ≠ m.end(); ++it){
  cout << "chiave:" << it→first << " valore:" << it→second;
}

pos = ma.find(1);
if (pos≠ma.end()){ m.erase (pos); } // verificare prima esistenza della chiave

if (m.count("NN")>0){ } // verifico l'esistenza di un elemento

map<list<int>,int>::iterator mliter;
for(mliter= mli.begin(); mliter≠mli.end();mliter++){
  //(mliter→first()).push_back(3);  NON PERMESSO MODIFICARE CHIAVE
}
```

9.0.1.8. Multimap

Contenute in . Funzionano esattamente come le map, tranne che si possono avere più elementi con la stessa chiave.

9.0.1.9. Bitset

http://www.cplusplus.com/reference/bitset/
Consiste in un array di bit che viene memorizzato anche fisicamente come array di bit.

```
#include <bitset>
#include <sstream> // per istringstream

bitset<64> b;
b = bitset<64>(s);
cout << b.size(); // 64

// precendenza a operatori unari
cout << (b & (b << 3));
cout << (b | (b << 3));
cout << ~b

istringstream(s) >> b; // funziona come il costruttore

cout << b.to_ullong();
cout << b.to_ulong();

b ⟸ 4; // shift a sinista
b.flip(); // equivalente a: b = ~b;
b.reset(42); // resetta il 42esimo bit

// indicizzazione tramite operator[]
if (!b[42]) {
  b[41] = 0;
}
```

Metodi non visti a lezione, ma da approfondire:

```
count // numero di bit settati
test  // ritorna il valore del bit
any   // c'è qualche bit a uno?
none  // tutti i bit sono a zero?
all   // tutti i bit sono a uno?
set(posizione=tutti, valore=1)   // imposta tutti i b
to_string // ottieni la rappresentazione in stringa
```

`bitset::operator[]` restituisce un `std::bitset::reference` se applicato ad un oggetto bitset non const. Simula la referenza ad un bool.

```cpp
class bitset::reference {
  friend class bitset;
  reference() noexcept;
public:
  ~reference();
  operator bool() const noexcept; // convert to bool
  reference& operator= (bool x) noexcept; // assign bool
  reference& operator= (const reference& x) noexcept; // assign bit
  reference& flip() noexcept; // flip bit value
  bool operator~() const noexcept; // return inverse value
}

// b[5].flip(); dovrebbe funzionare
```

In alternativa possiamo utilizzare un `vector<bool>`. Al massimo abbiamo visto che si può usare il metodo `flip()`.

9.0.2. Iterators

https://www.cplusplus.com/reference/iterator/
Strumento tramite cui si riesce ad accedere agli elementi all'interno di un contenitore, ma senza estrarli (simili ai puntatori, anche se non lo sono).
Sono una gerarchia di template. Dal più scarno al più ricco di funzionalità:

- input, output;
- forward;
- bidirectional;
- random access.

Il tipo di iteratore è determinato dal contenitore su cui lo usiamo:

- `set<A>::iterator` per insiemi;
- `multiset<A>::iterator` per multiset;
- `list<A>::iterator` per liste;
- `vector<A>::iterator` per vettori;
- `map<A>::iterator` per mappe.

Per tutti esiste l'operatore `++`.

Nota su l'iteratore `.end()`: si riferisce ad un elemento che **segue** l'ultimo elemento della lista. Dunque end non si può mai dereferenziare.

Numero di elementi fra due iteratori: `distance(v.begin(), found)`.

Esistono i `const_iterator` che non permettono di modificare gli elementi (`*cit = 3` errore a compile time).

```cpp
// B ha un campo list<A*> lpa;
// dato che il metodo è const allora dobbiamo usare un const_iterator
void B::stampa() const {
  list<A*>::const_iterator it;
  for (it = lpa.cbegin(); it != lpa.cend(); ++it){
    (*it)->stampa(); // doppia dereferenziazione
  }
}
```

```cpp
ostream&  operator<<(ostream& os, const cineteca& _c){
  // deve necessariamente essere del tipo const_iterator perché
  // stiamo lavorando sulla cineteca, che è costante
  map<int, film>::const_iterator it;
  for (it = _c.m.begin(); it != _c.m.end(); it++){
    // accede al primo e al secondo elemento del paio
    os << it->first <<  " "  << it->second <<  " ";
  }
  return os;
}
```

Per visitare al contrario un container possiamo usare i `reverse_iterator` e i `const_reverse_iterator`:

```cpp
list<int>::reverse_iterator rit;
for (rit = l.rbegin(); rit != l.rend(); ++rit){
  cout << *rit;
}
```

Inoltre possiamo usare dal C++11 anche la sintassi `auto`:

```cpp
int dato[10];
for (auto valore : dato) {}

vector v = {0, 1, 2, 3, 4, 5};
for (auto n : v) { }  // non modificabili
for (auto& n : v) { } // ref modificabili
for (const auto& n : v) { } // const ref non modificabili

// valido solo dal c++17 -> evita le copie
for (auto& n : as_const(v)) { } // usare se in sola lettura
```

9.0.3. Algorithms

https://www.cplusplus.com/reference/algorithm/
Sono algoritmi che cercano di essere i più generici possibili rispetto al container, basati sull'iteratore più semplice possibile. Agli algoritmi infatti passiamo gli iteratori e non i container.

```cpp
// Nota le funzioni sono passate con: &nomefunz
// (online sembra funzionare anche senza &)
#include <algorithm>
all_of(first, last, item -> bool)   -> bool
any_of (first, last, item -> bool)  -> bool
none_of(first, last, item -> bool)  -> bool
for_each(first, last, item -> _) // come for : auto, ma con estremi
for_each(execution::par, first, last, item -> _) // da C++17
find(first, last, item) -> iterator // primo elemento; usa operator=
find_if(first,last,item->bool)->it //prima occorrenza; last if not found
count_if(first, last, item -> bool) -> int // in realtà non è int
mismatch(first1,last1,first2,(item,item) -> bool) -> pair(ptr1, ptr2)
equal(first1, last1, first2, [(item, item) -> bool]) -> bool
```

```
is_permutation(first1, last1, first2) -> bool // at most n^2
replace(first, last, old_item, new_item)
remove(first, last, item) -> iterator // removes all occurences
reverse(first, last) // in place
rotate(first, middle, last) // in place

// posizione successiva a ultimo elemento copiato
copy(first1, last1, first2) -> iterator
#include <iterator>
copy(vi.begin(), vi.end(), ostream_iterator<int>(cout, " "));

transform(first1,last1,first2, f:x->y)//in first2 il risultato di f(x)
transform(first1, last1, first2, f:(x,y)->z)//prende x da 1 e y da 2

sort(first, last, [(item, item) -> bool]) // in place
sort(rbegin, rend) // ordine decrescente

#include <random>
int myrandom (int i) { return rand()%i; }
shuffle(first, last, default_random_engine()); //distrub. uniforme
shuffle(first, last, myrandom); // in place
shuffle(first, last, mt19937{random_device{}()});//non deterministici

accumulate(first,last,int)->int//somma elementi e li aggiunge al numero
accumulate(first, last, int, (int, int) -> int) -> int // (acc, item)

#include <numeric>
// inizializza con numeri crescenti dall'intero specificato
iota(first, last, int)
```

10. Multithreading

Classe thread introdotta dal C++11

I thread sono dei sotto-processi che possono essere eseguiti in parallelo o in serie. Essi collaborano per il raggiungimento di uno scopo comune e, per questo motivo, possono anche condividere le stesse risorse. Ci permettono di scrivere programmi più efficienti in quanto sfruttiamo tutti i core.

Possiamo passare diverse cose ad un thread:

- funzioni libere;
- funzioni membro;
- oggetti functor;
- espressioni lambda.

```
class Strana {
  public:
    void stampa(int dato){ cout << dato << endl; }
    void operator()(int val) { cout << val << endl; }
};
void stampa(int dato){ cout << dato << endl; }

```

```
int main(){
  thread t1(stampa, 9); // funzione libera

  Strano molto;
  thread t2(&Strana::stampa, &molto, 10); // funzione membro

  thread threadClass(moltoStrano, 10); // oggetto functor

  auto mylambdaExp = [](int a) { cout << a << endl; };
  thread mythread(mylambdaExp, 10); // lambda expression
}
```

Questo esempio espone il problema del *data race*: non è possibile accedere ad una variabile comune da più thread, in quanto il risultato della computazione è indefinito. Esistono diverse soluzioni:

- rendere atomiche le operazioni su una certa variabile con `<atomic>` (`template <class T> struct atomic`);
- definizione di sezioni critiche mutualmente esclusive tramite oggetti bloccanti `<mutex>`; ricordiamo però che il mutex non è in grado di risolvere tutti i problemi di sincronizzazione.

```
#include <thread>
#include <atomic>

// static int condivisa = 0;
static std::atomic<int> condivisa(0);

void inc_thread(){ for(int i=0; i<10000; i++) condivisa++; }
void dec_thread(){ for (int i=0; i<10000; i++) condivisa--; }

int main() {
  unsigned int c = std::thread::hardware_concurrency();
  std::cout << " numero di core: " << c << std::endl;

  // avvia i thread
  // nota: non è dato sapere chi inizia prima
  std::thread inc(inc_thread);
  std::thread dec(dec_thread);

  // sincronizza threads
  inc.join(); // pausa attende il primo
  dec.join(); // pausa attende il secondo
  std::cout << "condivisa:" << condivisa;

  // L'accesso in memoria condivisa può avvenire in qualsiasi momento
  // 1) Con var statica probabilmente il risultato non è 0
  //    questo perché ++ e -- non sono operazioni atomiche
  // 2) Usando atomic nessun problema: C++ gestisce l'atomicità
  //     delle operazioni sulla variabile (solo per i tipi base)
  return 0;
}
```

```
#include <thread>
```

```cpp
#include <mutex>

std::mutex mtx; // mutex sezione critica
void print_block (int n, char c) {
  // sezione critica accesso esclusivo a std::cout
  mtx.lock();
  for (int i=0; i<n; ++i){ std::cout << c; }
  std::cout << '\n';
  mtx.unlock();
}

std::mutex g_display_mutex;
void thread_function() {
  g_display_mutex.lock();
  std::thread::id this_id = std::this_thread::get_id();
  std::cout << "thread id: " << this_id << std::endl;
  g_display_mutex.unlock(); // non eseguita se c'è un'eccezione
}

std::mutex g_display_mutex2;
void thread_function2() {
  // inizializza il mutex a lock; quando esce dallo scope fa
  // l'unlock, garantito anche in caso di eccezioni
  std::lock_guard<std::mutex> guard(g_display_mutex2);
  std::thread::id this_id = std::this_thread::get_id();
  std::cout << "sono il thread " << this_id << std::endl;
}

std::mutex myMutex;
static int condivisa=0; // non serve atomic, gestiamo coi mutex

// overhead di un/lockare ogni iterazione è molto costoso
void inc_thread(){
  for (int i=0; i<10000; i++){
    myMutex.lock();
    condivisa++;
    myMutex.unlock();
  }
}
void dec_thread(){
  for (int i=0; i<10000; i++){
    myMutex.lock();
    condivisa--;
    myMutex.unlock();
  }
}

int main (){
  std::thread th1 (print_block,50,'*');
  std::thread th2 (print_block,50,'$');
  th1.join(); th2.join();

  std::thread t1(thread_function);
  std::thread t2(thread_function);
  std::thread t3(thread_function);
```

```
  std::thread t4(thread_function);
  t1.join(); t2.join(); t3.join(); t4.join();

  std::thread T1(thread_function2);
  std::thread T2(thread_function2);
  std::thread T3(thread_function2);
  std::thread T4(thread_function2);
  T1.join(); T2.join(); T3.join(); T4.join();

  std::thread inc(inc_thread);
  std::thread dec(dec_thread);
  inc.join(); dec.join();
  std::cout << "condivisa" << condivisa;
  return 0;
}
```

Alcuni problemi della concorrenza:

- *deadlock*: situazione in cui uno o più processi/thread si bloccano. Può avere diverse cause:
  - omissione di unlock del mutex
  - terminazione inattesa di una funzione (lancio di eccezioni): soluzione `lock_guard<mutex>`
  - funzioni annidate che chiamano lo stesso mutex: soluzione `recursive_mutex`
  - ordine di locking di diversi mutex: soluzione `lock(mutex, mutex)`
- *starvation*: la politica di accesso impedisce ad un processo pronto di accedere alla risorsa che necessita per essere eseguito
  Nella letteratura esistono semafori e monitor, ma non vengono implementati dal C++ perché preferisce il basso livello.

11. Eccezioni

http://www.cplusplus.com/reference/stdexcept/
Il meccanismo delle eccezioni prevede che se un'eccezione sollevata in una funzione che non è in grado di gestirla, allora viene propagata all'indietro verso il chiamante, risalendo lo stack delle chiamate. Nel caso dovesse risalire fino al main (e non dovesse essere gestita) allora il programma termina.

Elenco delle più usate:

- `throw invalid_argument("valore negativo");`
- `throw overflow_error("divisione b=0!");`
- `throw out_of_range("a o b maggiori di 9");`
- `throw logic_error("grosso guaio");`
- `throw bad_alloc("memoria finita");`

```
#include <stdexcept>
// #include <cstdlib>
// #include <string>
try {
  int x, y;
  cin >> x >> y;
  if (y==0) { throw "Divisione per 0!"; }
} catch (const char* messaggio){
```

```
    cerr << messaggio << endl;
}

try {
    try { //se qualcosa va storto lancio un'eccezione
        throw logic_error("grosso guaio");
        throw int(3);
        throw double(3.3);
    } catch(int i){
        cout << "intercettata eccezione" << i;
    } catch(const logic_error& e){
        cout << e.what();
        // la catch può rilanciare a sua volta
        throw; // rilancia la STESSA eccezione
    } catch(...){
        cout << "intercettata eccezione sconosciuta";
    }
} catch(logic_error e){
    cout << "try esterno:" << e.what();
}
```

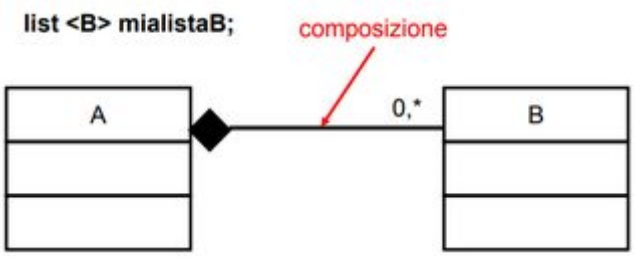
```
try {
    // cerchiamo di far finire la memoria
    while(true){ int* p=new int[99999999]; }
}
// l'ordine conta: la bad_alloc è una exception
//catch(exception& e){cout<<e.what();}
catch(bad_alloc a){
    cout << "memoria finita programma termina" << a.what();
} catch(exception& e){
    cout << e.what();
} catch(...){
    cout << "Qualcosa è andato storto";
}
```

12. Relazioni tra classi

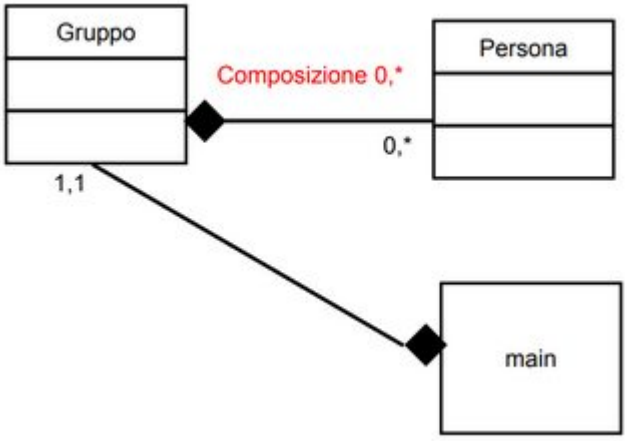
Losanga:

- vuota → aggregazione
- piena → composizione

12.0.1. Composizione



```cpp
class B {
  private:
    int dato;
  public:
    B(int _dato){ dato=_dato; }
    ~B(){ }
    B(const B& b){ dato = b.dato; }
    void stampa() const { }
};

class A {
 private:
   list<B> listb;   //losanga piena (lista di istanze)
   // list<B*> listpb; losanga vuota (riferimenti ad istanze)
 public:
   A(){ }
   ~A(){ }
   void addB(int dato){ B b(dato); listb.push_back(b); }
   void stampa(){ } // non mettere const, altrimenti usa const_iterator
   friend ostream& operator <<(ostream &os, A& a);
};

int main(int argc, char *argv[]){
  A a; a.addB(5); a.addB(2); a.addB(9);
}
```

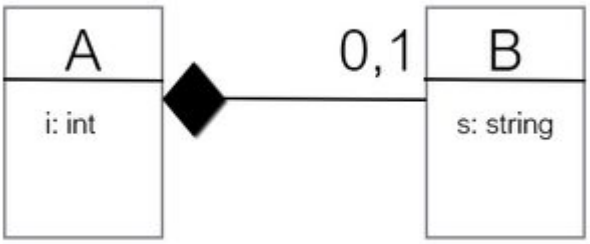


La corrispettiva aggregazione deve Persona collegata al main.

```cpp
class Persona {
  private:
    string nome;
  public:
    Persona();
    Persona(string n);
    Persona(const Persona& p);
    ~Persona();
    bool operator<(const Persona& p) const;
};
class Gruppo {
  private:
    set<Persona> sp; // composizione 0,*
```

```
    // set<Persona*> spp; aggregazione 0,*
  public:
    Gruppo();
    ~Gruppo();
    // 1 e 2 sbagliati perché il main non deve conoscere Persona
    //void addPersona1(Persona p);
    //void addPersona2(const Persona& p);

    void addPersona3(string nome); // main non conosce persona, OK
    // void addPersona(Persona *pp);  // main conosce persona (aggreg.)
    void remPersona3(const string& p); // usa la find_if
    void remPersona(const Persona& p); // usa la find
};
int main(int argc, char** argv) {
  Gruppo g; Persona p1("Mario"); Persona p2;
  g.addPersona(&p1);
  g.remPersona(p1);
}
```

Composizione opzionale 0,1 (=facoltativa)



La classe A ha la responsabilità di gestire la memoria in cui si trova l'oggetto di tipo B.

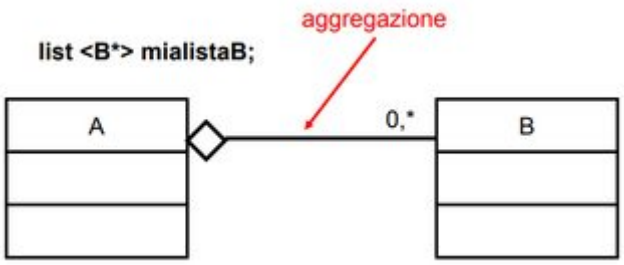
```
class B {
  string s;
  public:
    B(string _s);
    string get_s(); // getter
};

class A {
  int i;
  B* pb; // puntatore composizione opzionale 0,1
  public:
    A() { i=0; pb=NULL; }
    A(int _i, string _s) { i=_i; pb=new B(_s); }
    // copy constructor (profonda)
    A(const A& _a) {
      i = _a.i;
      if (_a.pb != NULL) { pb = new B(*(_a.pb)); }
      else { pb = NULL; }
    }
    //a=(b=c) operatore di assegnazione (4 casi)
    A&  operator=(const A& _a){
      if (this->pb == NULL) {
        if (_a.pb != NULL) { pb = new B(*(_a.pb)); }
      } else {
```

```
        if (_a.pb ≠ NULL) { (*pb) = *(_a.pb); }
        else { delete pb; pb = NULL; }
      }
    }
    ~A() { if (pb ≠ NULL) { delete pb; } }
    int get_i(); // getter
    string get_s(); // getter con controllo se pb ≠ NULL
    void set_s(string _s) {
      if (pb == NULL) { pb = new B(_s); }
      else { (*pb) = B(_s); } // costruttore di assegnazione
    }
};
```

12.0.2. Aggregazione

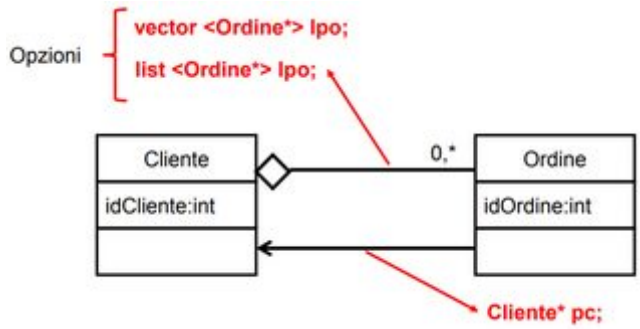
Cardinalità 1,*

```
#include <list>

class A {
  int i;
  public:
    A(int _i);
};

class B {
  // list<A> la; NO, questa è una composizione
  list<A*> lpa;
  public:
    B(A& _a);
    void add(A& _a);
};
```

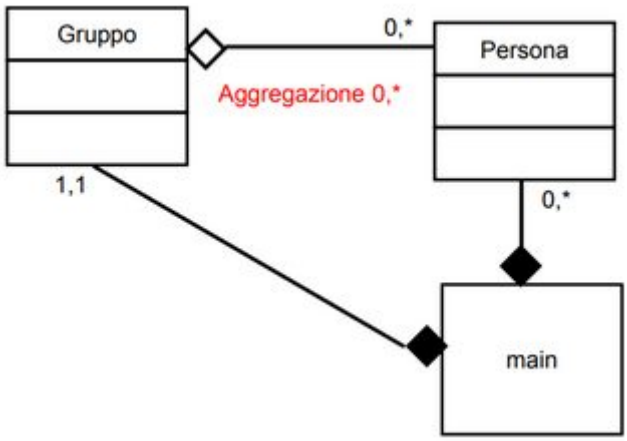



```
class A;
class B {
 private:
  int dato;
 public:
   B(int _dato){ dato = _dato; }
   ~B(){ }
   B(const B& b){ dato = b.dato; }
   void stampa() const { }
  friend ostream& operator <<(ostream &os, B& b);
};
```

```
12
13  class A {
14   private:
15     list<B*> listpb; // losanga vuota (riferimenti ad istanze)
16   public:
17     A(){ }
18     ~A(){ }
19     void addB(B* b){ listpb.push_back(b); }
20     void stampa(){} //no const con iterator, altrimenti const_iterator
21     friend ostream& operator <<(ostream &os, A& a);
22  };
23
24  int main(int argc, char *argv[]){
25    B b1(5); B b2(2); B b3(9);
26    A a; a.addB(&b1); a.addB(&b2); a.addB(&b3);
27  }
```

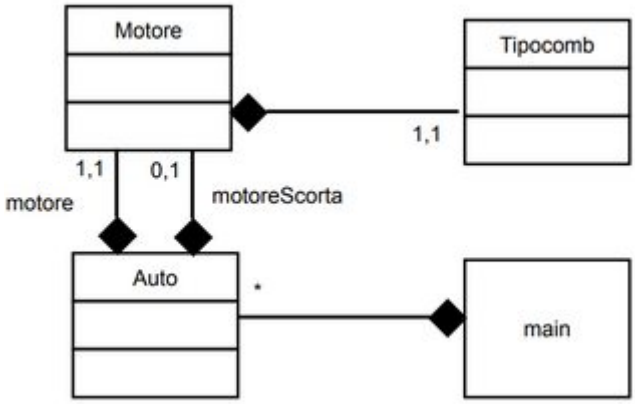



```
1   class Ordine;
2   class Cliente {
3     private:
4       int idCliente;
5       list<Ordine*> lpo;
6     public:
7       Cliente(int id){ idCliente = id; }
8       ~Cliente(){ }
9       void addOrdine(Ordine* po){ lpo.push_front(po); }
10  };
11  class Ordine{
12    private:
13      int idOrdine;
14      Cliente *pc;
15    public:
16      Ordine(int id){ idOrdine = id; pc = NULL; }
17      ~Ordine(){ }
18      void setCliente(Cliente* c){ pc = c; }
19  };
20  int main(int argc, char** argv) {
21    Ordine o1(1), o2(2), o3(3);
22    Cliente c1(1);
23    c1.addOrdine(&o1); c1.addOrdine(&o2); c1.addOrdine(&o3);
24  }
```




```cpp
class Persona {
  private:
    int idPersona;
    string nome;
  public:
    Persona(){}
    Persona(string n, int id);
    ~Persona();
    int get_id() const;
};

class Gruppo {
  private:
    // map<int, Persona> mp; composizione 0,*
    map<int, Persona*> mpp; // aggregazione 0,*
  public:
    // non devo passare nessun parametro in quanto sia aggr. che composizione
    // hanno molteplicità 0. Sarebbe stato diverso con composizione pari a 1
    Gruppo();
    ~Gruppo();
    Persona search(int id);

    // void addPersona(Persona p);  per copia: DA EVITARE
    // void addPersona1(const Persona& p); per riferimento: DA EVITARE

    // per la composizione: Persona non collegata al main
    // void addPersona2(string nome, int id); param per creare una Persona

    void addPersona3(Persona* pp) { // per indirizzo (aggregazione)
      mpp.insert(pair<int, Persona*>(pp->get_id(), pp));
    }
};

int main(int argc, char** argv) {
  Persona p1("Mario", 234); Persona p2("Luca", 999);
  Gruppo g;
  g.addPersona(p1); g.addPersona1(p2);
  g.addPersona3(&p2);
  Persona p = g.search(999); cout << p << endl;
}
```

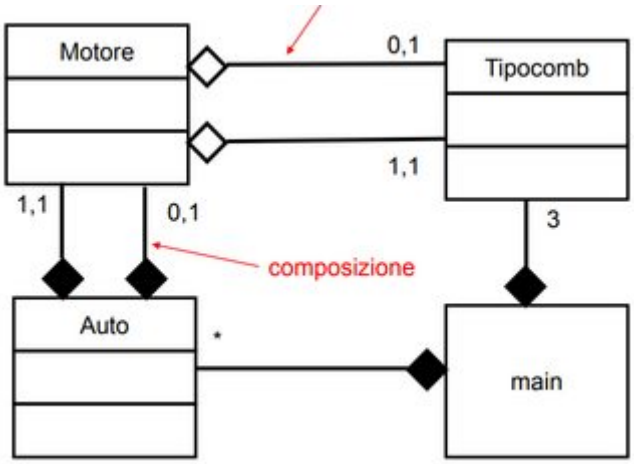
12.0.3. Has-A




```
int main(int argc, char *argv[]){
  Auto ferrari(BENZINA, 3000,"ferrari","testarossa");
  ferrari.setMotoreScorta(DIESEL,2000);
}

class Auto {
  private:
    char* marca; char* modello;
    Motore motore;
    Motore* motoreScorta;
  public:
    Auto(Tcombustibile _tipo,int _cil,char* _marca, char* _modello);
    ~Auto();
    Auto(const Auto& a);
    void setMotoreScorta(Tcombustibile _tipo, int _cil){
      // mancano controlli se fosse già assegnato
      motoreScorta = new Motore(_tipo, _cil);
    }
};

class Motore {
  private:
    int cil;
    Tipocomb tipo;
  public:
    Motore(Tipocomb _tipocomb,int _cil);
    Motore(Tcombustibile _tipo,int _cil=1000);
    ~Motore();
    int get_cil() const;
    Tipocomb get_tipo() const;
};

typedef enum {DIESEL, BENZINA, GPL} Tcombustibile;
class Tipocomb {
  private:
    Tcombustibile comb;
  public:
    Tipocomb(Tcombustibile _comb);
    ~Tipocomb();
```

```
};
```

12.0.4. Use-A




```
int main(int argc, char *argv[]){
  Tipocomb combBENZINA(BENZINA);
  Tipocomb combDIESEL(DIESEL);
  Tipocomb combGPL(GPL);
  Tipocomb *comb = new Tipocomb(GPL);

  Auto auto1(&combBENZINA, 1100, "Fiat","Panda");
  auto1.addMotoretipo(&combGPL);
  Auto auto2(comb,3500,"Ferrari","Enzo");
  auto2.setMotscorta(&combDIESEL,2000);
}

class Auto {
  private:
    Motore motore;  //losanga piena 1,1
    Motore* motscorta;//losanga piena 0,1
    string marca; string modello;
  public:
    // con _tipo e _cil settiamo motore; motscorta a NULL
    Auto(Tipocomb* _tipo, int _cil,string _ma, string _mo);
    ~Auto();
    void setMotscorta(Tipocomb* _tipo, int _cil){
      // mancano controlli se fosse già assegnato
      motoreScorta = new Motore(_tipo, _cil);
    }
    // serve solo per passare a motore il combustibile opzionale
    void addMotoretipo(Tipocomb* _tipo){
      motore.setTipo2(_tipo);
    }
};

class Motore {
  private:
    Tipocomb* tipo; //losanga vuota 1,1
    Tipocomb* tipo2; //losanga vuota 0,1
    int cilindrata;
```

```
  public:
    Motore(Tipocomb* _tipo, int _cil);
    ~Motore();
    void setTipo2(Tipocomb* _tipo) { tipo2 = tipo; }
};

typedef enum {DIESEL, BENZINA, GPL} Tcombustibile;
class Tipocomb {
  private:
    Tcombustibile comb;
  public:
    Tipocomb(Tcombustibile _comb);
    ~Tipocomb();
};

```

13. Template Metaprogramming

Per *metaprogramma* si intende qualsiasi programma che ha come input o output un altro programma (es. compilatore).
La computazione tramite template avviene definendo i tipi in modo ricorsivo.

Punti chiave che permettono il funzionamento di questi meccanismi:

- le espressioni vengono calcolate al tempo di compilazione (preferire `constexpr` per fare questa cosa);
- laziness, vengono istanziati solo i tipi che si usano;
- specializzazione dei template (permette di definire il caso base per terminare la ricorsione);
- deduzione dei tipi dei template: avvia tutto il meccanismo in fase di compilazione;
- Non si possono fare assegnazioni! Un tipo è definito per sempre, come nei linguaggi funzionali.

Utilità della meta programmazione:

- è possibile spostare computazione da runtime a compile time
- si possono generare tipi che dipendono dall'hardware
- si possono generare tipo specializzati più efficienti.

Turing completezza:

- controllo condizionale e ricorsione (al massimo qualche migliaio per ora);
- forniamo una prova "formale" che permette di generare macchine di Turing a stati (e di conseguenza anche una macchina di Turing universale)

	Runtime functional program	C++ template metaprogramming
values	run-time data (constant, literal)	static const and enum class members
variables	variables	symbolic names (typename, typedefs)
initialization	constants generators	static const initialization, enum definition
assignment	no	no
i/o helpers	monads	warnings, error messages, no interactive input
branching	pattern matching function specialization	pattern matching template specialization
looping	recursive functions	recursive templates
subprogram	function	(template) class
data types	abstract data structures	typelists, boost::vector
types	type class (Haskell)	concepts

```cpp
#include <iostream>

template<unsigned int n> // caso induttivo
struct factorial {
  enum { value=n*factorial<n-1>::value };
};

template<> // caso base
struct factorial<0>{
  enum { value=1 };
};

int main(){
  // verranno definiti a compile-time i fattoriali da 7 in giù
  cout << factorial<7>::value;
}
```

13.0.1. Controllo IF

```cpp
#include <iostream>

template <bool condition, class Then, class Else>
struct IF {
  typedef Then RET;
};

// caso particolare del template di sopra
template <class Then, class Else>
```

```
struct IF<false, Then, Else> {
  typedef Else RET;
};

int main(){
  // definizione diversa a tempo di compilazione in base a sizeof
  IF< sizeof(int)<sizeof(long), long, int >::RET i;
  cout << sizeof(i) << endl;
}
```

13.0.2. max_IF

```
#include <iostream>

template <bool condition, class Then, class Else>
struct IF {
  typedef Then RET;
};

template <class Then, class Else>
struct IF<false, Then, Else> {
  typedef Else RET;
};

// la deduzione di parametri avviene in fase di compilazione
// dunque molto efficiente a runtime
template <class T1, class T2>
typename IF< sizeof(T1)<sizeof(T2), T2, T1 >::RET max(T1 a, T2 b){
  return a > b ? a: b;
}

int main(){
  // qui non ci sono problemi con il tipo del risultato
  cout << max(2, 3);
  cout << max(3, 2.3);
  cout << max(int(2), double(3.14));
  cout << sizeof(double);
}
```

13.0.3. Accumulate

```
// Accumulate(n, f) := f(0) + f(1) + ... + f(n)

template<int n, template<int> class F>
struct Accumulate {
  enum { RET = Accumulate<n-1,F>::RET + F<n>::RET };
};

template <template<int> class F>
struct Accumulate<0,F> {
  enum { RET = F<0>::RET };
};
```

```
12
13 template <int n> // funzione su cui eseguire l'accumulate
14 struct Square {
15   enum { RET = n*n };
16 };
17
18 int main(){
19   cout << Accumulate<3, Square>::RET;
20 }
```