Programmazione Avanzata

Dispensa riassuntiva anno acc. 2020/2021

0. Basi C++

In questo capitolo verranno raccolte nozioni e informazioni varie riguardanti concetti spiegati durante il corso che non ricadono necessariamente in uno degli altri argomenti, ma potrebbero essere necessarie per la comprensione degli stessi.

0.1. Metodi e operatori di default

Nella definizione standard di una classe in C++, anche se completamente vuota, sono definiti di default alcuni metodi e operatori necessari al funzionamento di base di una classe.

Costruttore vuoto

```
Classe::Classe();
```

Permette di creare una istanza della classe, occupando quindi la memoria necessaria, ma senza eseguire altre operazioni (come inizializzare eventuali attributi).
Questo costruttore viene utilizzato solo se non ne è stato definito un qualsiasi altro dall'utente, con o senza parametri.

Costruttore di copia

```
Classe::Classe(const Classe& c);
```

Questo corruttore permette di creare una nuova istanza della classe copiando gli attributi della istanza passata come parametro.
Viene utilizzato dal costruttore di copia di default e nel passaggio di parametri per valore.

Operatore d'assegnamento

```
Classe& Classe::operator=(const Classe& c);
```

Permette di copiare i valori di una classe all'interno di un'altra istanza tramite il costruttore di copia.

Distruttore

```
Classe::~Classe();
```

Il distruttore è un metodo particolare che viene invocato quando la istanza di un oggetto viene eliminata dalla memoria, nel caso venga usato delete o semplicemente lo scope di cui faceva parte è terminato.
La sua versione di default non fa nulla di particolare.

0.2. Passaggio di parametri

In C++ esistono 3 metodi per passare dei parametri a funzioni o metodi:

Per copia (per valore)

```
void funzione(Classe c);
```

In questo caso viene utilizzato il costruttore di copia della classe (o tipo) per creare una nuova istanza del parametro da utilizzare all'interno della funzione. Qualsiasi modifica apportata al parametro non influirà sulla sua istanza esterna alla funzione.

Per indirizzo

```
void funzione(Classe *c);
```

In questo caso la funzione richiede che le venga passato un puntatore della classe (o tipo) indicata, di conseguenza all'interno della funzione il parametro dovrà essere trattato come un puntatore e dovrà essere dereferenziato o utilizzato con l'operatore ->.
In quando il parametro farà riferimento allo stesso spazio di memoria del valore all'esterno della funzione, qualsiasi modifica effettuata all'interno della funzione si ripercuoterà all'esterno.

Per riferimento

```
void funzione(Classe &c);
```

Nel passaggio di parametri per riferimento invece, la funzione richiede che venga passato la effettiva istanza della classe (o tipo) e non un riferimento o puntatore, quindi all'interno della funzione il parametro verrà utilizzato come

se fosse un passaggio per copia (per valore), tuttavia come nel passaggio per riferimento qualsiasi modifica apportata all'interno della funzione si ripercuoterà anche all'esterno.

Const

La parola chiave const viene utilizzata per specificare che un parametro passato ad una funzione deve rimanere costante, e quindi ne viene impedita la modifica all'interno della funzione.
Const torna utile con il passaggio per indirizzo o per riferimento di un parametro che non deve essere modificato, in questo modo non si corrono rischi e si ha il vantaggio di non dover eseguire una operazione di copia, in certi casi molto dispendiosa.

```
void funzione(const Classe *c);
void funzione(const Classe &c);
```

0.3. Costruttori

Come è stato detto nella sezione 0.1. in una classe viene definito in automatico un costruttore senza parametri, è ovviamente possibile definire un qualsiasi tipo e quantità di costruttori in base alle esigenze.
Ricordiamo che la definizione di un qualsiasi tipo di costruttore elimina il costruttore di default.

Costruttori con parametri

In un costruttore possono essere specificati quanti parametri necessari per popolare una istanza della classe o per eseguire qualsiasi altra operazione.
Prendiamo come esempio una semplice classe A con due attributi.

```
class A{
     private:
          int e;
          int v;
     public:
          A(int _e, int _v){
               e = _e;
               v = _v;
          };
};
```

Nel costruttore definito all'interno di A vengono passati due parametri di tipo int e semplicemente vengono inizializzati gli attributi della istanza.
Ovviamente non è necessario inizializzare tutti gli attributi di una classe per crearne una istanza, e si possono creare più costruttori con un numero diverso di parametri per eseguire operazioni diverse.
Inoltre è possibile specificare dei valori di default per i parametri, i quali verranno utilizzati nel caso il determinato parametro non venga specificato nella chiamata del costruttore.

```
A(int _e = 1, int _v){
     e = _e;
     v = _v;
};
```

Nel costruttore definito sopra il parametro _e ha come valore di default 1, quindi nel caso non venisse specificato non sarebbe un problema, tuttavia _v deve essere specificato obbligatoriamente.
Inoltre è possibile inizializzare un attributo ancora prima che la istanza venga inizializzata (la istanza è inizializzata dall'inizio del costruttore).

```
A(int _e, int v):e(_e){
     v = _v;
};
```

In questo modo l'attributo e viene inizializzato con il contenuto di _e, però l'operazione di assegnamento non è stata eseguita successivamente alla creazione della istanza, questo metodo permette di assegnare dei valori arbitrari ad attributi costanti, altrimenti impossibili da modificare una volta creati.

Costruttori con un parametro

I costruttori con un solo parametro ricoprono una funzione particolare in C++, permettono la conversione implicita dei tipi.
Prendiamo come esempio questa semplice classe:

```
class A{
     int e;
     public:
          A(int _e){e = _e;};
};
```

Con la classe A appena definita sono valide le seguenti espressioni:

```
void fun(A a){…};

A a = 3;
fun(5);
//presupponendo che l'operatore + sia stato definito
A b = a + 7;
```

I casi sopra esposti vengono cos' interpretati dal compilatore:

```
void fun(A a){…};

A a = A(3);
fun(A(5));
//presupponendo che l'operatore + sia stato definito
A b = a + A(7);
```

Pur essendo una funzione comoda in molti casi è possibile che in certe situazioni la si voglia evitare, a questo scopo esiste la parola chiave explicit, la quale disabilita la conversione automatica per il costruttore a cui è applicata e costringe a utilizzarlo esplicitamente.

```
class A{
     int e;
     public:
          explicit A(int _e){e = _e;};
};
```

1. Caratteristiche avanzate C++

1.1. Operator overloading

Data la possibilità di definire nuovi tipi (classi) oltre a quelli naturali del C, si rende necessario anche poter definire il comportamento degli operatori rispetto a questi tipi (classi), possibilità comunque comune nei linguaggi moderni che utilizzano la OOP.
A livello pratico gli operatori hanno un determinato tipo di ritorno e determinati parametri, possono essere definiti come metodi della classe o come funzioni esterne (meno eccezioni).

Di seguito una tabella con gli operatori di cui si è discusso di più a lezione:

Operatore	Metodo interno	Funzione esterna
a = b	`A& A::operator=(T b);`	`--------`
a + b , a - b, a * b, a / b	`A A::operator+(T b);`	`A operator+(A a, T b);`
a += b , a -=b , a *= b , a /= b	`A& A::operator+=(T b);`	`A& operator+=(A& a, TB);`
++a , —a	`A& A::operator++();`	`A& operator++(A& a);`
a++ , a—	`A A::operator++(int);`	`A operator++(A& a, int);`
a == b, a != b	`bool A::operator==(T const& b) const;`	`bool operator==(A const& a, T const& b);`
a > b , a < b, a >= b . a <= b	`bool A::operator==(T const& b) const;`	`bool operator==(A const& a, T const& b);`
b << a	`idk`	`ostream& operator<<(ostream& os, A const& a);`

Operatore d’assegnazione

```
A& A::operator=(A& b);
A& A::operator=(A b);
A& A::operator=(int b);
```

L’operatore d’assegnazione è l’unico visto a lezione che può essere definito solo come metodo interno alla classe, di conseguenza, come tutti gli altri metodi interni, la istanza chiamante (quella a cui sta venendo assegnato qualcosa, l’operando a sinistra per intenderci) è direttamente accessibile nel corpo del metodo.
Il valore di ritorno è ciò che verrà assegnato alla istanza chiamante (tecnicamente), il parametro rappresenta ciò che sta venendo assegnato, può essere di qualsiasi tipo o classe, in base alle esigenze e implementazioni.

Operatori aritmetici

```
R A::operator+(T b); R operator+(A& a, T b);
R A::operator-(T b); R operator-(A& a, T b);
R A::operator*(T b); R operator*(A& a, T b);
R A::operator/(T b); R operator/(A& a, T b);
```

Gli operatori aritmetici sono abbastanza intuitivi, l’operando a sinistra è la istanza chiamante, che nel caso della implementazione come metodo interno sarà accessibile direttamente, mentre nella implementazione come funzione esterna sarà rappresentata dal primo parametro (generalmente passato per riferimento, ma non dovrebbe essere obbligatorio, ed è comunque consigliabile targare entrambi i parametri con const).
Il valore di ritorno è ovviamente il valore risultante dalla operazione, che come si può notare non è per forza dello stesso tipo della classe chiamante.

Operatori di incremento e decremento post e pre fissi

```
Prefissi
A& A::operator++(); A& operator++(A& a);
A& A::operator--(); A& operator--(A& a);
Postfissi
A A::operator++(int); A operator++(A& a, int);
A A::operator--(int); A operator--(A& a, int);
```

Gli operatori di incremento e decremento sono quelli un po' più particolari, in quanto è necessario distinguere tra post e pre fisso.
A livello di firma della funzione, sia che sia un metodo interno o funzione esterna, gli operatori postfissi hanno un parametro int "dummy", inoltre il tipo di ritorno dei prefissi deve essere per forza un puntatore alla istanza chiamante.
A livello implementato gli operatori prefissi eseguono prima la operazione sulla istanza e poi ne ritornano un riferimento, mentre i postfissi devono ritornare la istanza chiamante allo stato prima della operazione, per questo la prima cosa da fare all'interno della funzione e salvarsi una copia della istanza.
In entrambi i casi l'operatore andrà a modificare la istanza della classe, quindi non può essere segnata come const.

Operatore di shift logico

```
ostream& operator<<(ostream& os, const A& a);
```

Durante le lezioni l'operatore di shift a sinistra viene utilizzato per implementare un metodo di stampa più naturale, usando la funzione cout, permette inoltre di concatenare più oggetti o stringhe (stream in ogni caso), quindi è stata definita come sopra, in teoria si può definire come metodo interno e i tipi di ritorno e del primo parametro non sono per forza ostream.
Per questa implementazione è necessario includere la libreria <iostream> per poter accedere al tipo ostream.
A livello logico il corpo della funzione dovrebbe creare una rappresentazione sotto forma di stringa di A e concatenarla ad os riutilizzando l'operatore <<.
In realtà, se definito così, l'operatore non è propriamente una ridefinizione dell'operatore di A, bensì di ostream, aggiungendo la possibilità di concatenarci A.

Parola chiave friend

Per poter rendere possibili certe operazioni ridefinendo gli operatori come funzioni esterne potrebbe essere necessario targarli con il modificatore friend all'interno della definizione della classe.
Questo operatore segnala alla classe che quella determinata funzione ha la possibilità di accedere ai suoi attributi privati come se fosse un metodo interno.

1.2. Copia profonda

Nel capitolo introduttivo si era parlato del costruttore di copia, il quale permette svariate operazioni, come il passaggio di parametri per valore o l'assegnamento standard.
Tuttavia la sua implementazione di default utilizza la così detta copia superficiale.

Copia superficiale

La copia superficiale consiste nel semplice copiare il contenuto di tutti gli attributi di una istanza e copiarli in una istanza diversa.
Questo tipo di operazione molto spesso non è un problema, ma può causare comportamenti sgradevoli nel caso una classe abbia come attributo un puntatore.

```
class A {
public: int n;
};

class B {
     int v;
     A* a;
};
```

Date le classi appena definite, se avessimo una istanza di B (con tutti i parametri inizializzati), e provassimo ad assegnarla ad un'altra istanza di B, ci ritroveremmo con due istanze di una classe che puntano ad una sola istanza di A, in quanto con la copia superficiale si sarebbe semplicemente copiato il contenuto dell'attributo a (che è un indirizzo).
In questi casi è necessario ridefinire il costruttore di copia in modo che esegua una copia profonda.

Copia profonda

La copia profonda non fa altro che controllare più a fondo il contenuto degli attributi di una classe e nel caso di riferimenti ad altre classi si preoccupa di creare nuove istanze da assegnare alla istanza di destinazione.

```
class A {
public: int n;
};

class B {
     int v;
     A* a;
     public:
          B(const B& b){
               v = b.v;
               if(b.a != null){
                    a = A();
                    a.n = b.a->n;
               } else {
                    a = null;
               }
          };
};
```

1.3. Ereditarietà

L'ereditarietà è un meccanismo proprio dei linguaggi che adottano il paradigma ad oggetti.
Permette di dare ad una classe tutti gli attributi e metodi di un'altra ma con la possibilità di aggiungerne o modificarne alcuni (detto in termini poveri).
Questa funzione ha dei vantaggi che vedremo in seguito.

Ereditarietà in generale

Esistono 3 tipi di ereditarietà e fanno riferimento alle keywords di visibilità degli attributi in una classe: Public, Private, Protected.
Il tipo di ereditarietà utilizzata modificherà di conseguenza la visibilità degli attributi e metodi per la classe che eredita.

	Public	**Protected**	**Private**
Public	Public	Protected	-
Protected	Protected	Protected	-
Private	Private	Private	-

Le righe della tabella rappresentano i tipi di ereditarietà, mentre le colonne sono i tipi di visibilità degli attributi, nel caso la visibilità sia private gli attributi saranno completamente inaccessibili.

Una caratteristica importante della ereditarietà, e anche una delle più utili, è che una istanza di una classe derivata di tipo public (figlia) può essere acceduta come istanza della classe base (padre), in particolare si può utilizzare un puntatore della classe base per puntare una istanza della classe derivata, e si può assegnare una istanza della classe derivata ad una variabile del tipo della classe base.

```
class A{};

class B: public A{};

void fun(A a);

A a = B();
A* a1 = new B();
fun(B());
```

Costruttori della classi derivate

Una particolarità da tenere presente quando si eredita da una classe è che la classe figlia contiene gli attributi di quella base pur non essendo specificati esplicitamente, di conseguenza quando si definisce il costruttore si vorrà inizializzare questi attributi, il problema è che potrebbero non essere accessibili.
Per risolvere questo dilemma diventa necessario richiamare il costruttore della classe padre prima del corpo del costruttore della classe figlio.

```
class A{
     int i;

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


class B{
     public:
          B(int _i):A(_i){};
};
```

Keyword virtual

La keyword virtual può essere utilizzata per definire un metodo di una classe come tale, nel qual caso verrà utilizzato il late binding quando viene richiamato.
La differenza tra Late e Early binding è che l'early viene risolto durante la compilazione, quindi si sa dal principio qual'è l'istanza chiamante, mentre il late viene risolto a run time.
Importante notare che il late binding viene applicato solo se si accede tramite puntatore o riferimento.
Diventa necessario definire come virtual un metodo che può avere implementazioni differenti in base a quale classe si sta utilizzando, nello specifico, se si rende necessario utilizzare una variabile di un tipo di una classe base per contenere una istanza di una classe derivata si rende necessario definire il distruttore come virtual, se no la sua esecuzione diverrebbe errata.

```
class A{
     public:
          int i;
          virtual void fun(){};
          virtual ~A(){};
};

class B: public A{
     public:
          ~B(){};
};
```

Classe puramente virtuale

Una classe puramente virtuale è una classe che contiene almeno un metodo puramente virtuale.
Un metodo puramente virtuale è un metodo virtuale non implementato che nella sua dichiarazione ha un "=0".

```
class A{
     public:
          virtual void func()=0;
}:
```

La particolarità di una classe puramente virtuale è che non può essere istanziata.
Lo scopo di una classe puramente virtuale è ovviamente essere ereditata così da implementare i metodi completamente virtuali, il risultato è un insieme di classi che ereditano da una stessa classe virtuale (e quindi possono essere identificati dallo stesso tipo di variabile) ma con implementazioni diverse.

Ereditarietà multipla

Oltre a far ereditare da una classe padre a più classe figlie è possibile far ereditare a una classe figlia da più classi padri.

```
class A {};

class B {};

class C: public A, public B{};
```

Da notare che l'ordine con cui vengono ereditate le classi nel codice determina l'ordine con cui le classi vengono create.
Una problematica della ereditarietà multipla si ha nel momento in cui le classi da cui si eredita hanno attributi o metodi uguali, rendendo ambigua la invocazione da parte di una istanza della classe figlia.
Un modo diretto per risolvere il problema è specificare lo scope del metodo che si vuole usare, se proprio si vuole evitare di ridefinire il metodo.

```
class A{
     public:
          void fun(){};
};

class B{
     public:
          void fun(){};
};

class C: public A, public B{};

C c;
c.A::fun();
c.B::fun();
```

Problema del diamante

Il problema del diamante è una situazione particolare con la eredità multipla, succede quando si hanno due classi che ereditano da una stessa classe padre, e una quarta classe che eredita dalle prime due.
La problematica sta nel fatto che quando si va ad inizializzare la quarta classe verranno create due istanze della prima tramite la seconda e la terza, quando dovrebbe essere solo una, a livello logico almeno.

```
class A{};

class B: public A{};

class C: public A{};

class D: public B, public C{};
```

Per risolvere questa situazione bisogna utilizzare la keyword virtual nella ereditarietà delle classi di mezzo, tuttavia questo metodo ci costringe a definire un costruttore senza parametri nella prima classe.

```
class A{
     public:
          A(){};
};

class B: virtual public A{};

class C: virtual public A{};

class D: public B, public C{};
```

1.4. Eccezioni

Le eccezioni sono un costrutto che permette di intercettare una serie di errori logici che possono capitare durante la esecuzione di un programma, evitando così il blocco della esecuzione e permettendo di gestire l'errore.
Il C++, come altri linguaggi ad oggetti, permette di "lanciare" le eccezioni (throw) e di "catturarle" con il costrutto try - catch.
Il comando throw in particolare, che è quello che ci permette di segnalare gli errori, va a creare un oggetto di una classe particolare in base all'errore che si è generato.

Dentro il blocco try si inseriscono le istruzioni che rischiano di lanciare errori, per tenerle sotto controllo per così dire.
Un blocco try è sempre seguito da uno o più blocchi catch, ai quali è specificato a quale errore fanno riferimento.
Quindi se durante la esecuzione di un blocco try si alza una eccezione verrà fermata l'esecuzione delle istruzioni di quel blocco e verrà avviato il blocco catch adeguato alla eccezione avvenuta.

```
void fun(){
     throw errore;
};

try{
     //istruzioni che possono dare errori
     fun();

} catch (errore) {
     //gestione della eccezione
}
```

Il tipo di eccezione sollevata può differire, con le eccezioni generali è possibile anche passare al catch una stringa come descrizione.

```
try{
     throw "errore";
} catch (const char* mix){
     cout << mex << endl;
}
```

Libreria <stdexcept>

Per poter sollevare eccezioni di tipo diverso da quelle testuali viste poco fa è necessario utilizzare la libreria stdexcept, assieme al tipo si eccezione si può inoltre passare un messaggio testuale.
Di conseguenza si possono definire dei catch specifici per tipo di eccezione, specificando come parametro un riferimento al tipo di eccezione desiderata.
Nel caso ci sia bisogno di un catch generale, che non è legato ad una eccezione specifica, si segna come parametro "...".
Inoltre bisogna prestare attenzione all'ordine con cui si definiscono i catch, in quanto vengono controllati in ordine, viene quindi richiamato il primo catch adeguato.
Qui la pagina di riferimento per stdexcept:
https://www.cplusplus.com/reference/stdexcept/

1.5. Standard Template Library e Iterators

Le STL o Standard Template Library sono un insieme di librerie che definiscono dei tipi (contenitori) che fanno utilizzo dei template per l'utilizzo di un approccio di programmazione generica.

Template e programmazione generica

```
template <class T>
class TempClass{

};
```

Per programmazione generica ci si riferisce alla possibilità di scrivere algoritmi che abbiano un tipo come parametro, permettendo così di astrarre certe cose di un programma.
Il tipo parametrico verrà poi specificato a tempo di compilazione in C++, grazie all'utilizzo dei template (in altri linguaggi, come java, vengono specificati a run time).
Utilizzando la programmazione generica si possono implementare degli algoritmi che sono indipendenti dal tipo dei parametri.
Nell'esempio appena fatto si definisce una funzione swap che utilizza un template per astrarre i parametri e permettere l'esecuzione con qualsiasi tipo di variabile.
Per permettere ciò si definisce un template prima della funzione, assegnandogli un "segnaposto" (in questo caso T), dopodiché nella chiamata della funzione viene specificato il tipo da utilizzare (nella prima <int> e nella seconda <string>) al posto di T.

```
template <typename T>
void swap(T& a, T& b){
     T tmp = a;
     a = b;
     b = tmp;
};

int a = 2, b = 3;
string s1 = "str1", s2 = "str2";

swap<int>(a, b);
swap<string>(s1,s2);
```

Ovviamente è possibile definire più di un template, i quali possono far riferimento anche a due tipi diversi.
Inoltre i template non sono applicabili solo a funzioni, ma anche a classi utilizzando la parola chiave class dichiarando i tipi template.

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

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

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;
};
```

Contenitori

I contenitori sono delle classi facenti parte della STL che sono caratterizzate dal poter “contenere” dei dati di un determinato tipo parametrico.

Vector

- Memoria contigua.
- Pre-alloca spazio per elementi futuri.
- Ogni elemento occupa spazio solo per il tipo dello stesso.
- Ogni volta che viene aggiunto un elemento può riallocare tutto il vettore.
- Inserimenti alla fine sono di costo costante, mentre in qualsiasi altra posizione hanno peso O(n).

- Stessa cosa per la cancellazione di un elemento.
- Accesso randomico con [ ].
- Gli iteratori si invalidano se viene tolto o inserito un elemento.
- Si può trasformare in un array.

List

- Memoria non contigua.
- No memoria preallocata, il memory overhead è costante.
- Ogni elemento richiede spazio anche per i puntatori all'elemento successivo e precedente.
- Non ha mai bisogno di riallocare tutta la lista quando si aggiunge un elemento.
- Inserimento e cancellazione di elementi hanno un costo computazionale basso in qualsiasi caso.
- Non si può accedere agli elementi in modo randomico, quindi l'accesso può essere costoso.
- Gli iteratori rimangono validi anche dopo aver modificato la lista.
- Creare un array da una lista richiede una operazione manuale.

Set

- La ricerca di un elemento ha costo logaritmico sulla dimensione.
- L'inserimento e la cancellazione sono logaritmici in generale.
- Gli elementi sono sempre ordinati dal più basso al più alto, utilizzando l'oggetto di confronto interno del tipo parametrico (che deve quindi essere per forza definito).
- Ogni elemento è unico.
- Gli elementi di un set non possono essere modificati (sono const), solo inseriti o rimossi.

Unordered Set

A differenza dei set normali, non sono ordinati con l'operazione di confronto, sono invece organizzati in sottoinsiemi determinati dal valore hash degli elementi.
Quindi gli unordered set sono più veloci nell'accesso dei singoli elementi tramite chiave, ma meno efficienti nella iterazione in un intervallo.

Multi set

A differenza dei set normali, i multi seti permettono di inserire più volte dei valori uguali, sono uguali per il resto.

Unordered Multi Set

Come gli unordered set gli elementi non sono ordinati ma organizzati in

category				properties	valid expressions
all categories				*copy-constructible*, *copy-assignable* and *destructible*	`X b(a);` `b = a;`
				Can be incremented	`++a` `a++`
Random Access	Bidirectional	Forward	Input	Supports equality/inequality comparisons	`a == b` `a != b`
				Can be dereferenced as an *rvalue*	`*a` `a->m`
			Output	Can be dereferenced as an *lvalue* (only for *mutable iterator types*)	`*a = t` `*a++ = t`
				default-constructible	`X a;` `X()`
				Multi-pass: neither dereferencing nor incrementing affects dereferenceability	`{ b=a; *a++;` `*b; }`

				Can be decremented	`--a`
					`a--`
					`*a--`

				Supports arithmetic operators + and –	`a + n`
					`n + a`
					`a - n`
					`a - b`

				Supports inequality comparisons (<, >, <= and >=) between iterators	`a < b`
					`a > b`
					`a <= b`
					`a >= b`

				Supports compound assignment operations += and -=	`a += n` `a -= n`
				Supports offset dereference operator ([])	`a[n]`

sottoinsiemi determinati dal valore hash, oltretutto gli elementi non sono unici.

Map

- Contenitore di dimensioni variabili, efficiente nel recuperare i valori degli elementi associati ad una chiave.
- Ha iteratori bidirezionali.
- Ordinati per il valore delle chiavi (le quali devono avere la funzione di confronto definita).
- Le chiavi sono univoche.

- Contenitore associativo, dato che ha un valore chiave e valore mappato.
- Permettono l'accesso diretto ai valori tramite l'operatore [ ] e la chiave.

Iteratori

```
#include <list>

list<int> L = list<int>(10,0);//lista inizializzata con 10 zeri
list<int>::iterator it;

for(it=L.begin(); it!=L.end(), it++){
     //operazioni sui valori accessibili da *it
}
```

Sono un insieme di classi template che riproducono in parte il funzionamento dei puntatori, pur non essendolo e ridefinendo degli operatori tipici dei puntatori.
A livello pratico gli iteratori permettono di accedere ai valori di un contenitore senza dover estrarre il valore stesso.
Come è ovvio dal nome degli iteratori questi vengono utilizzati per iterare i valori di un contenitore.
Esistono più classi di operatori, ognuna con una serie di operatori ridefiniti, in generale però ogni iteratore può essere incrementato (a++, ++a, in modo di spostarsi tra gli elementi del contenitore).
Da notare che la tipologia di iteratore utilizzabile è determinata dal tipo di contenitore che si sta utilizzando, in quanto tutti gli iteratori non sono supportati da tutti i contenitori.

Algorithm

La libreria Algorithm contiene funzioni basate su iteratori, in modo da essere il più astratti possibili, essendo generici anche rispetto ai containers.
https://www.cplusplus.com/reference/algorithm/

Puntatori a funzione

```
int (*pfunz)(double , int);
int f1(double, int);
pfunz = f1;
(*pfunz)(2.0,3);
pfunz(2.0,3);
```

I puntatori a funzione sono molto banalmente dei puntatori a determinati tipi di funzione, in base alla dichiarazione del puntatore.
Come si può vedere dall'esempio sopra, il puntatore pfunz può contenere un tipo di funzione che prende come parametri un double e un int e restituisce un int, dopodiché può essere richiamato usando la dereferenziazione o no.
I puntatori a funzione possono essere passati come parametri ad altre funzioni, permettendo la definizione di algoritmi più generali, con parti di essi che possono essere fornite tramite parametro.
Molto spesso vengono utilizzati nella libreria algorithm.

Oggetti funzione

Gli oggetti funzione sono oggetti a cui è stato ridefinito l'operatore ( ).
La ridefinizione dell'operatore ( ) permette di utilizzare istanze di questi oggetti come se fossero funzioni, da tener presente che la ridefinizione di questo operatore è molto libera, avendo un numero e tipo di parametri indefinito, si può quindi ridefinire più volte.

1.6. Namespaces

I namespace sono un insieme di identificativi creati dal programmatore per definire degli scope per evitare il conflitto tra nomi di funzioni e classi.

2. Tecniche moderne C++11

2.1. Auto e structured binding

La parola chiave auto permette di dichiarare variabili il cui tipo viene dedotto dal compilatore in base al contesto, attenzione che non permette di avere una variabile tipizzata dinamicamente (come in python), la variabile manterrà il suo tipo per tutta la esecuzione, ma rimanda la tipizzazione al compilatore invece che al programmatore.
Una delle utilità di auto viste a lezione è la inizializzazione automatica di un iterator per un ciclo su un contenitore.

```
vector<int> s = vector<int>(10,0);

for(auto it = s.begin(); it!=s.end(); ++it){
     //codice
}
```

La variabile it utilizzata per iterare su s viene tipizzata dal compilatore in base a cosa gli viene assegnato, nell'esempio sopra viene usato il metodo begin della classe vector, il quale ritorna un iteratore all'inizio della istanza.

2.2. Range-based Loops

I range-based loops sono una nuova sintassi per definire un ciclo for, hanno esattamente la stessa funzione, ma sono più comodi e leggibili.
La sintassi è la seguente:

```
vector<int> s = vector<int>(10,0);

for(int i : s){
     //codice
}
```

La variabile i rappresenta i valori contenuti in s, acquisisce il valore di un item alla volta per ogni iterazione del ciclo.
Ovviamente è possibile dichiarare la variabile iteratice con auto, in modo da non preoccuparsi del tipo.

2.3. Rvalues e move semantic

Oggetti temporanei

Per capire l'utilità della move semantica è innanzitutto necessario sapere dell'esistenza degli oggetti temporanei e delle loro problematiche.
Gli oggetti temporanei vengono creati per restituire valori da funzioni o per valorizzare una espressione con operatori la cui funzione di overload deve essere valutata.
Il problema risiede nel fatto che lo scopo di questi oggetti è essere copiato in altri oggetti, appesantendo l'esecuzione.

Lvalues e Rvalues

Semplicemente gli Lvalues sono i valori che possono stare a sinistra di un operatore di assegnazione, mentre gli Rvalues sono quelli che possono stare a destra.
In realtà esistono più categorie oltre a queste due.

Rvalues references

```
int&& i = 7;
```

Una rvalue reference (semantica come sopra) permette di definire un riferimento ad un valore rvalue, tutto ciò senza effettuare una copia.

Move

La funzione move è una funzione template che prende in input un rvalue e ne

```
int j = 1;
int i = move(j++);
```

restituisce una reference.
Nell'esempio sopra la funzione prende in input il valore temporaneo creato da j++, e lo riassegna a i.
Il vantaggio di questa funzione è che permette di evitare di copiare i valori, cosa che sarebbe avvenuta se si fosse semplicemente assegnato il valore a i.

Move con classi

Per permettere l'utilizzo della move anche con delle classi definite dal programmatore bisogna ridefinire il costruttore move e il move operator, tenendo presente che la istanza di riferimento passata come parametro a questi due metodi deve essere "annullata" dopo averla trasferita.

```
class A{
     public:
          A(A&& _a);
          A& operator=(A&& _a);
};
```

2.4. Smart/Unique/Shared pointers

Smart pointers

Gli smart pointers sono delle classi template che creano dei puntatori con una gestione della memoria "furba".
Sono definiti nella libreria <memory>.
Mantengono le proprietà dei puntatori normali, per la maggior parte, ma la loro eliminazione è automatizzata, non c'è infatti bisogno di eliminarli manualmente, verranno automaticamente distrutti alla fine dello scope.
Anche se questo funzionamento assomiglia a quello di un garage collector di altri linguaggi, non lo è, l'eliminazione del puntatore avviene seguendo le regole d'ambito di C++, infatti è il distruttore del puntatore intelligente a contenere il codice per eliminare la risorsa dall'heap (inoltre è da tener presente che il puntatore intelligente si trova nello stack).

Unique pointer

```
unique_ptr<int> p1, p2;
p1.reset(new int(54));
cout << *p1;
p2 = move(p);
```

Unique pointer è un tipo di smart pointer che non può essere copiato in alcun modo, utilizza principalmente la move semantica per le sue operazioni.
Inoltre eliminano immediatamente l'oggetto a cui puntano nel momento in cui loro stessi vengono distrutti.

Come si può vedere dall'esempio sopra, un puntatore di questo tipo si inizializza con il metodo rese, dopodiché lo si può usare come un normale puntatore, meno che per l'assegnazione, che ci costringe ad utilizzare move.
Unique pointer, dovendo contenere informazioni solo del suo puntatore, ha effettivamente le stesse dimensioni di un puntatore normale, rendendolo molto efficiente e una buona alternativa ai puntatori classici.

Shared pointer

Con questa classe si ha più istanze che condividono la proprietà di una risorsa che viene deallocata solo quando l'ultimo puntatore viene deallocato.
Si può usare la funzione use_count per sapere quanti sono gli owner.

Weak pointer

Permette di condividere una risorsa ma senza averne l'ownership, quindi senza incrementare use_count.
Per accedere al contenuto bisogna copiarlo in uno shared_ptr usando la funzione lock.

2.5. Lambda expression

Le espressioni lambda sono una sorta di funzione senza nome, vengono definite la dove vengono usate.
Il fatto che non abbiano un nome le caratterizza in quanto sono temporanee, non possono essere richiamate da altre parti del programma.
Il punto essenziale delle lambda espressione sta nella cattura delle variabili nel contesto in cui sono.

```
int i = 2;
auto x = [i](int j) -> int { return i+j;};
x(3) // 5
```

La sintassi delle espressioni lambda è come segue: la prima parte tra parentesi quadre definisce quali variabili verranno catturate per essere utilizzate nel corpo, tra parentesi tonde si definiscono eventuali parametri, si usa l'operatore -> per definire il tipo di ritorno e tra parentesi graffe si implementa il corpo della funzione.
Con la parola chiave mutable è possibile rendere mutabili le variabili catturate dalla espressione lambda, le quali sono costanti in altri casi.

```
[i, j] () mutable -> void { j = i+2;};
```

Le variabili catturate possono essere passate sia per copia che per riferimento.

```
[&i] () {//codice};
```

Inoltre è possibile omettere certi elementi dalla sintassi, quali il tipo di ritorno (nel qual caso sarà dedotto) e la lista di parametri (nel qual caso sarà considerata vuota), o entrambi.

```
[i] (int j) -> int {//codice};
[i] (int j) {//codice};
[i] {//codice};
```

2.6. Metaprogramming

Con metaprogramma si intende un qualsiasi programma che ha come input o output un altro programma (per esempio i compilatori).

Casi particolari nei template

Per poter proseguire con la metaprogrammazione è necessario spiegare qualche concetto in più riguardo ai template.
Finora è stato visto che i template possono fungere come generalizzazione, però è possibile anche specificare una restrizione sul tipo o specializzare funzioni e strutture in base a che valore viene passato come tipo template.

```
template <int n>
struct fact{ enum {v=n*fact<n-1>::v};};

template<>
struct fact<0>{ enum {v = 1;};};

cout << fact<3>::v;
```

Nell'esempio sopra viene definita la struttura fact che rappresenta un fattoriale, in primo luogo viene specificato che il valore template deve essere

per forza un int, e dopo viene definita una versione particolare nel caso quel valore sia 0.
Inoltre in questo esempio viene mostrata un'altra caratteristica importante dei template, ovvero la definizione ricorsiva, si può notare infatti come una istanza di fact sia definita in base ad un'altra istanza di fact con n-1.

Strutture di controllo in metaprogrammazione

```
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;
};

IF<sizeof(int)<sizeof(long) ,long ,int>::RET i;
```

L'esempio sopra definisce una struttura di controllo IF, il funzionamento pratico è che restituisce un tipo in base a se la condizione è vera o falsa, si può vedere come nella applicazione si decide il tipo della variabile i (tramite il typedef RET) in base a se int è più piccolo o meno di long.
Considerare che l'esempio funziona in quanto viene risolto a tempo di compilazione, come tutti i casi con i template.
Il funzionamento può sembrare un po' complesso all'inizio, vengono definite due strutture IF, una generale e una specializzata, entrambe prendono come parametri template una condizione booleana e due tipi qualsiasi, tuttavia viene utilizzata la prima struttura (che ritorna il primo tipo generico) solo se la condizione è vera, in quanto la seconda specifica che la condizione è falsa, e verrà utilizzata sempre in quel caso.

Funzioni in metaprogrammazione

```
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};
};

template <int n>
struct Square{
     enum {RET=n*n};
};

Accumulate<3,Square>::RET;
```

L'esempio sopra definisce una struttura funzione che ha lo scopo di accumulare (sommare) i risultati della funzione che gli viene passata su un insieme di valori.
I valori template usati in questo caso sono : un intero n che indica i valori da usare (da 0 a n), e un altro valore template<int> che rappresenta la funzione che viene passata (ovviamente essendo metaprogrammazione, sarà un'altra struttura funzione).
La implementazione specifica utilizza 0 come caso base di n, quindi notiamo come la implementazione generale si definisce ricorsivamente sommando i valori risultanti della funzione template con parametro n.
Essendo poi la funzione passata per parametro una funzione square avremo che l'esempio accumula i valori quadrati dei numeri da 0 a 3, quindi: 3*3 + 2*2 + 1*1 + 0*0 = 14.

3. Multithread

Il multithreading è la possibilità di un programma di eseguire più processi che condividono lo stesso spazio di memoria contemporaneamente.

Thread

La libreria <thread> di C++11 mette a disposizione la classe thread, che permette di inizializzare e gestire un processo.

```
void f1(){…};
void f2(int i){…};

thread f (f1);
thread s (f2,0);

f.join();
s.join();
```

Per costruire un thread bisogna definirne una variabile passando al suo costruttore un “callable” (in poche parole una funzione), è possibile specificare dei parametri come altri argomenti, tenendo presente che la funzione deve essere sempre il primo parametro.

```
int cond=0;
mutex m;

void up(){ m.lock(); cond++; m.unlock();};
void do(){ m.lock(); cond--; m.unlock();};

thread f(up);
thread s(do);

f.join();
s.join();
```

Si utilizza la funzione join per mettere in pausa il processo finché la istanza da cui è chiamato termina.

Data race

La data race è una condizione in cui due processi (thread) differenti operano (almeno uno in scrittura) sullo stesso dato, ciò può portare a risultati

inaspettati in quanto non si può prevedere che operazione viene eseguita per prima.

Atomic

Utilizzare la libreria <atomic> per rendere le istruzioni di un thread atomiche è una delle soluzioni alla data race.
La classe atomic è una template class che garantisce l'atomicità delle operazioni eseguite sul tipo specificato.
La classe funziona solo sui tipi base.

```
atomic<int> cond(0);

void up(){ cond++;};
void do(){ cond--;};

thread f(up);
thread s(do);

f.join();
s.join();
```

Come si può vedere dall'esempio la classe atomic ridefinisce gli operatori del tipo passato come parametro in modo che la sua istanza possa essere utilizzata come se fosse una istanza del tipo base.
Tuttavia non la si può riassegnare.

Mutex

La seconda soluzione al data race è rendere una variabile esclusiva all'utilizzo di un solo thread alla volta, ovvero si individuano nel codice delle sezioni chiamate sezioni critiche in cui è necessario sincronizzare l'accesso alle risorse per garantire la mutua esclusione.
Per far ciò C++ mette a disposizione la libreria <mutex>.
Una istanza di mutex permette di eseguire il lock e unlock per definire delle sezioni critiche in cui le risorse sono di utilizzo esclusivo del thread.