Università degli Studi di Napoli
Federico II

A.A 2023-2024

LASD

Indice

Introduzione

Estratto

Il seguente documento rappresenta una raccolta di appunti, riorganizzata e migliorata nelle vesti grafiche, del corso di LASD del docente Fabio Mogavero, per il corso di laurea in Informatica A.A 2023-2024, della Federico II di Napoli.

Si tiene a specificare che questo documento non ha lo scopo di **sostituire libri di testo**, note del professore o lezioni frontali, nonostante siano tutte fonti utilizzate per la stesura, ma quello di integrare al materiale più complesso una rilettura degli argomenti da un punto di vista meno tecnico e quindi più lento nel soffermarsi nei passaggi critici.

Si ringraziano **Valentino Bocchetti** e **Pasquale Miranda** ideatori e gestori del progetto *Unina Docs*,

Si ringraziano inoltre tutti i revisori che hanno contribuito al documento con consigli e proposte di modifica, invitando a dare feedback in caso di errori.

Si ringraziano in fine i lettori nella speranza che questo lavoro sia stato utile.

Redazione a cura di **Giorgio Di Fusco**, **Valentino Bocchetti**

Documento aggiornato al **16 aprile 2024** (Revisione documento **bfaf49d**)

Introduzione al linguaggio C++

1.1 La struttura di un programma

Il linguaggio C++ è un linguaggio compilato. Ciò vuol dire che un programma, per essere eseguito, necessita di essere tradotto da un compilatore che produce dei file oggetto i quali, combinati attraverso un **linker**, compongono un programma eseguibile. Un programma in C++ è composto tipicamente da vari codici sorgente come mostrato in Figura 1.1.

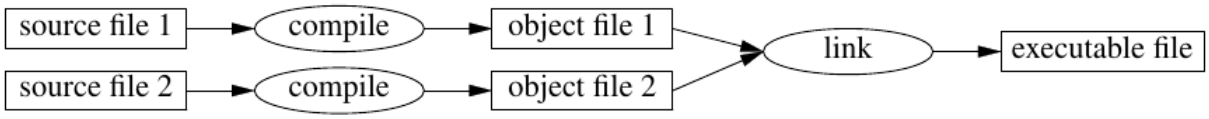


Figura 1.1: Struttura di un programma C++

```cpp
#include <iostream>
using namespace std;
int main()
{
  cout<<"Hello World!"<<endl;
  return 0;
}
```

Codice 1.1: Primo programma in C++: `main.cpp`

Da C99 e in C++ è possibile omettere il return nella funzione principale **main** se il flow del programma raggiunge la fine senza un valore di ritorno esplicito

Si consideri il Codice 1.1. Attraverso l'istruzione `#include <iostream>` viene importata la libreria `iostream` che è la corrispondente della Standard Library presente in C. Nella seconda riga viene dichiarato l'uso di un **namespace** chiamato `std`. Quando si hanno a che fare con numerose librerie è possibile che ci siano presenti funzioni omonime. Per evitare collisioni (*namespace pollution*) è stato quindi introdotto in C++ il concetto di `namespace`.

La sintassi generica per usare una funzione è la seguente:

```cpp
std::funzione
```

Specificando nel preambolo di un programma uno specifico `namespace` è possibile evitare questa sintassi che si rende però necessaria allorquando si utilizzino funzioni omonime appartenenti a librerie diverse.

! È sconsigliato includere l'intero namespace per evitare il polluting dell'ambiente (per altre good pratices si faccia riferimento al seguente video).

L'operatore `<<` ("put to") scrive il secondo argomento sul primo. In questo caso, la stringa letterale "Hello, World!" viene scritta sul flusso di output standard `std::cout`. Una stringa è una sequenza di caratteri circondata da doppi apici.

Per compilare `main.cpp` si può usare il comando g++ scrivendo: `g++ -o3 -o main.cpp` oppure inserendo il comando precedente all'interno di uno script Bash `build.sh`[1]:

[1] È consigliato per ovvi motivi sfruttare utility ad hoc per lo sviluppo come make e CMake

```
#!/bin/bash
g++ -O3 -o main.cpp
```

Anche se nell'esempio sovrastante viene utilizzato come livello di compilazione 3 (con questo valore si chiede al compilatore di ottimizzare il più possibile in favore delle performance), è spesso sconsigliato visto che in molti casi non si ha la garanzia di performance migliori, ma anzi si rischia di rallentare il sistema per binari di dimensioni maggiori e un incremento della memoria utilizzata. Il livello 2 è più che sufficiente.

1.1.1 Un programma un pelo più complicato

```
// main.cpp
#include <iostream>
#include"test.hpp"
using namespace std;
int main()
{
  cout<<"Hello World!"<<endl;
  test();
  return 0;
}
```

Codice 1.2: Disaccoppiare il programma in moduli

Dalla versione 20 è stato introdotto il concetto di modulo.

```
// test.hpp
#ifndef TEST_HPP
#define TEST_HPP
  void test();
#endif
```

Codice 1.3: Header file

```
// test.cpp
#include<iostream>
#include"test.hpp"
void test(){
  std::cout<<"OK!"<<std::endl;
}
```

Codice 1.4: Corpo della funzione test()

In questo esempio viene aggiunta una funzione di test all'interno del file `main.cpp`. Per farlo è preferibile separare il programma in moduli: un file contenente il `main`, un file per la definizione delle funzioni della libreria e un **header file** contenente i prototipi delle funzioni delle librerie.

Sfruttando il **linking** si compilano insieme i vari codici:

```
g++ -O2 -o main test.cpp HelloWorld.cpp
```

L'istruzione `INCLUDE` è detta **macro di precompilazione**, ovvero delle istruzioni date al **preprocessore** che, una volta lette le macro, prende i file specificati e incorpora il codice del file incluso. Per convenzione, i file di precompilazione sono scritti in maiuscolo. In questo senso l'*header file* mostrato nel Codice 1.3 funge da "ponte" tra le librerie e i programmi che ne usufruiscono come mostrato in Figura 1.2.

Le istruzioni `IFNDEF, DEFINE` vengono usate per evitare l'**importazione multipla** di librerie già importate. All'interno degli header file, infatti, non vanno inseriti i corpi della funzione e delle variabili. Questa è una cosa importante anche a livello di classi. A differenza di Java, dove non c'è separazione tra classi e il loro definizione, in C++ si usano gli header files per definire le classi mediante il costrutto `abstract` (come se fosse un'interfaccia Java) mentre l'implementazione della classe viene inserita in un apposito file `cpp`.

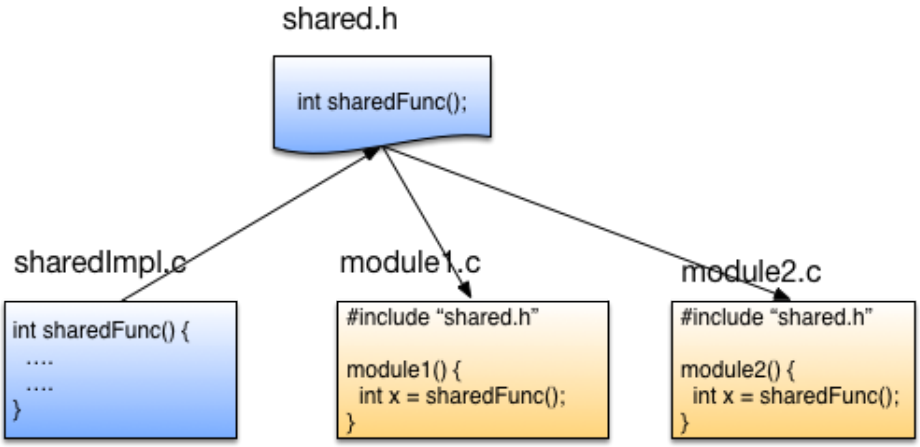


Figura 1.2: Utilizzo dell'header file per importare librerie

1.2 COME DICHIARARE UNA VARIABILE

Per dichiarare una variabile si usa la seguente sintassi:

```
type varname [= default value];
```

Mentre per dichiarare un array statico si usa la sintassi come segue:

```
type varname[num_elementi] [={val0, val1, ... , valn}];
```

dove al posto di `num_elementi` bisogna inserire un valore noto a tempo di compilazione, spesso una costante predefinita. Questo perché, quando si dichiara un array, questa viene inserita all'interno dello stack dove bisogna inserire *blocchi di dimensione fissata.*

1.2.1 Tipi fondamentali in C++

A differenza del C[2], in C++ è stato aggiunto il tipo `bool` di tipo booleano che accetta solo i valori `true` o `false`. Di default (per questioni di retrocompatibilità con il linguaggio C) quando si restituisce un valore di tipo booleano, `true=1` e `false=0`. Per visualizzare `true` e `false` basterà impostare il flag `std::boolalpha`.

```
#include <iostream>
int main() {
  std::cout<<false<<"\n"; // Stampa 0
  std::cout << std::boolalpha;
  std::cout<<false<<"\n"; // Stampa false
  return 0;
}
```

Per tutti i tipi di dato numerico esistono due versioni: **signed** o **unsigned**. Ad esempio, per quanto riguarda gli interi, ci sono 4 tipologie fondamentali come mostrato nella Tabella 1.1.

Tipo	Dimensione minima	Nota
`short int`	16 bit	
`int`	16 bit	Tipicamente 32bit su architetture moderne
`long int`	32 bit	
`long long int`	64 bit	

Tabella 1.1: Signed integers in C++

[2] A partire da C99 è stato aggiunto il supporto al tipo Boolean con il tipo built-in **_Bool**. Nel momento in cui è incluso l'header <stdbool.h> si ha accesso al tipo Boolean come bool.

Nonostante su alcune macchine alcuni tipi come `long` e `long long` possano avere lo stesso numero di bytes, questi vengono trattati dal linguaggio come tipi diversi per non avere effetti indesiderati su macchine con architetture differenti (i microprocessori moderni hanno normalmente parole di 16, 32 o 64 bit).

```
#include <iostream>
int main()
{
  short a;
  int b;
  long c;
  long long  d;
  std::cout<<"sizeof(short)="<<sizeof(a)*8<<std::endl;
  std::cout<<"sizeof(int)="<<sizeof(b)*8<<std::endl;
  std::cout<<"sizeof(long)="<<sizeof(c)*8<<std::endl;
  std::cout<<"sizeof(long long)="<<sizeof(d)*8<<std::endl;
  if(std::is_same<long,long long>::value)
    std::cout<<"They are the same"<<std::endl;
  else
    std::cout<<"They are not the same"<<std::endl;
  return 0;
}
```

Tipo	Dimensione
`bool`	1 byte
`char`	1 byte
`float`	4 byte
`double`	8 byte
`long double`	12 byte

Figura 1.3: Tipi fondamentali in C++

Per usare un `unsigned int` o un `unsigned long` è possibile usare la macro `uint` e `ulong`.

```
uint x = 3;
double vector[3]={0.4,2,5.7};
long matrix[3,2]={{0,1},{2,3},{4,5}};
```

Codice 1.5: Alcune dichiarazioni in C++

> **!** È buona prassi inizializzare sempre una variabile per facilitare il debugging ed evitare che eventuali dirty bit influenzino l'output.

1.2.2 Tipi costanti

Per dichiarare una costante si usa la keyword **`const`**:

```
const type varname = value;
```

Questa keyword si utilizza frequentemente sia nei parametri di funzione, sia come specificatore aggiuntivo delle funzioni. È utile dichiarare un parametro come `const` per evitare side effects dati dall'esecuzione della funzione. In questo modo si ottiene accesso in lettura ma non in scrittura.

1.2.3 Puntatori

Per una variabile di tipo `T`, una variabile di tipo `T*` è di tipo "puntatore a T". Per dichiarare un puntatore in C++ si usa la sintassi seguente:

```
type* varname [=default address];
```

Esempio

Nel Codice 1.6 è presente un esempio di dichiarazione di un puntatore in C++:

```
1 char c = 'a';
2 char* p = &c;
3 std::cout<<p<<endl;
4 p= nullptr; //nullptr corrisponde alla macro NULL del linguaggio C
5 std::cout << *p << endl;
```

Codice 1.6: Dichiarazione di un puntatore in C++

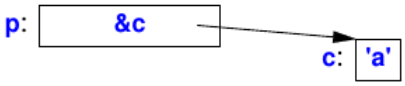


Figura 1.4: Puntatore

L'operazione principale per un puntatore è quella della **dereferenziazione**, ovvero l'accesso al valore dell'indirizzo di memoria puntata dal puntatore. L'operatore di dereferenziazione è il prefisso unario *. Ad esempio:

```
1 char c = 'a';
2 char* p = &c;
3 char c2 = *p;
```

Codice 1.7: Deferenziazione

Osservazione

Il tipo di un puntatore si legge da destra verso sinistra.

```
1 int* p;           //Puntatore ad un intero
2 char** pc;        //Puntatore ad un puntatore a char
3 int* ap[15];      //Array di 15 puntatori ad interi
4 int (*fp)(char*); //Puntatore ad una funzione
5 int* f(char*);    //Funzione che prende un parametro di tipo char* e restituisce un puntatore ad intero
```

Codice 1.8: Esempi di puntatori

1.2.4 Puntatori di tipo void

I puntatori a void sono dei puntatori di tipo "**naked**". Ovvero puntano a celle di memoria senza sapere il tipo di dato del valore che quella cella contiene. Questi puntatori sono utili soprattutto quando il programma non conosce il tipo di dato col quale si sta lavorando, utilissimo in contesti di dati astratti. Per dichiarare un puntatore a void si usa la seguente sintassi:

```
1 void* varname;
```

Non è possibile dereferenziare un puntatore void senza eseguire un cast. Esistono due modi per castare un puntatore void:

```
1 *((char*) p)
2 *(static_cast<char*>(p))
```

Non è possibile dereferenziare un void pointer e assegnarlo ad una variabile:

```
1 char d = *p; //Errore
2 char d = *((char*) p); //OK
```

Il linguaggio C++ mette a disposizione dello sviluppatore un metodo di indirizzamento della memoria alternativo all'uso dei puntatori: le variabili *reference*. I riferimenti non sono altro che degli **alias** per variabili già presenti all'interno della memoria locale del programma. La loro particolarità consiste nella modalità di accesso, che maschera l'operazione di dereferenziazione necessaria per leggere o scrivere un valore nella locazione di memoria contenuta in un normale puntatore.

A parte la loro definizione, infatti, per le variabili reference valgono le medesime regole sintattiche valide per l'accesso ad una variabile ordinaria. La differenza consiste nel fatto che una variabile reference non ha una locazione di memoria propria, ma punta a quella di una variabile ordinaria già esistente. Per questo motivo, l'inizializzazione deve avvenire contestualmente alla dichiarazione come mostrato nell'esempio seguente:

Esempio

Si consideri il Codice 1.9

```
1 unsigned int d = 7;
2 unsigned int &e = d;
3 e++;
4 cout<<d;
```

Codice 1.9: Primo uso delle reference

L'effetto di una operazione del genere sarà quella di aver incrementato di uno il valore della variabile `d`. A livello concreto è possibile che la cosa venga risolta per mezzo di puntatori. Quindi un compilatore, a meno che non sia specificato diversamente senza ottimizzazioni, eseguirebbe la seguente porzione di codice:

```
1 unsigned int d = 7; unsigned int* p = &d;
2 (*p)++; cout << d;
```

Per riflettere le distinzioni tra valori `lvalue, rvalue` e `const` o `non-const`, esistono tre tipi di riferimenti:

- **Riferimenti ad lvalue:** per ottenere un riferimenti agli oggetti dei quali vogliamo cambiare il valore;
- **Riferimenti const**: per riferirsi a oggetti il cui valore non si vuole cambiare;
- **Riferimenti ad rvalue:** per riferimenti ad oggetti i cui valori non devono essere preservati dopo il loro utilizzo.

Le prime due tipologie vanno sotto il nome di **riferimenti lvalue**.

Esempio

Si osservi la seguente istruzione:

```
1 int& i = 1; //Errore
```

Questa istruzione è un errore in quanto quello a destra è un valore costante e, per questo, non modificabile. Per ovviare a questo errore bisogna dichiarare un riferimento a valore costante:

```
1 const int& i = 1;
```

1.3.1 Riferimenti ad rvalue e operatore spostamento

`Rvalue` è sinonimo di una *espressione temporanea* che può essere salvata all'interno di un `lvalue`. Queste espressioni possono essere oggetti costanti come nel caso dell'esempio mostrato nel paragrafo precedente oppure valori restituiti da una funzione. I **riferimenti ad rvalue** sono riferimenti che vengono collegati solo ed esclusivamente ad oggetti temporanei.

```
1 std::string getName(){ return "Alex";}
2 int main(){
3   const string& name = getName(); // ok
4   string &name = getName(); // non ok
5 }
```

Un `lvalue reference` costante è in grado di estendere la vita dell'oggetto temporaneo, ma l'impossibilità nel modificarlo garantisce una sicurezza: rimarrebbe pericoloso memorizzare in esso una qualche informazione, visto che è destinato a scomparire. Dal C++11 è presente un nuovo tipo di riferimento, il **riferimento a rvalue**, il quale permette di collegare un riferimento non costante a un oggetto temporaneo, ma non a uno non temporaneo. In altre parole, gli `rvalue references` sono perfetti per rilevare la natura di un oggetto, ovvero determinare se esso è temporaneo o meno. Gli `rvalue references` vengono dichiarati tramite doppia e commerciale (`&&`) invece che una singola `&`, e possono essere costanti o non costanti, anche se raramente troveremo codice che utilizza un rvalue reference costante:

```
std::string getName(){ return "Alex";}
int main(){
  string &&name = getName(); // ok
  const string &&name = getName(); // ok
}
```

In che modo può esserci di aiuto tutto ciò? I riferimenti agli `rvalue` possono essere usati per implementare "letture distruttive" per l'ottimizzazione di ciò che altrimenti richiederebbe una copia.

```
T foo(T obj){
  T temp;
  return T;
}
```

Effettuando una chiamata alla funzione `foo` come segue: `T new = foo(old);` si effettua una copia, potenzialmente costosa, del valore di ritorno all'interno della variabile `new`. Grazie ai riferimenti `rvalue` è possibile implementare l'operatore `std:move`, chiamato **operatore di spostamento**, il quale altro non fa che fornire un riferimento all'oggetto temporaneo ed evitare che questo venga perduto o copiato una volta usciti dalla funzione chiamata: `move(x)` sta per `static_cast<X&&>(x)` dove `X` è il tipo di `x`.

```
#include <iostream>
string f(){ return string ("This is a long string"); }
int main(){
  string var = f(); //Copia della stringa
  string &&var = f(); //Riferimento temporaneo al valore di ritorno
  string &var = f();  //Errato
  string var1 = std::move(f());   //Spostamento del valore
  string var2 = std::move(var);   //Spostamento del valore
}
```

Codice 1.10: Riferimenti ad rvalue, copia e operatore move

1.4 ALLOCAZIONE DINAMICA DELLA MEMORIA

In C/C++ la gestione della memoria è *demandata al programmatore*. Come in C, in C++ esistono due principali operatori che permettono l'allocazione e la deallocazione della memoria: `new()` e `delete()`.

In caso di allocazione non andata a buon fine viene sollevata un'eccezione `bad_alloc` a differenza del linguaggio C dove è compito del programmatore verificare che non sia stato restituito un valore `NULL`. È buona norma quindi racchiudere le operazioni di allocazione dinamica all'interno di opportuni blocchi `try-catch`.

```
// Dichiarazione di un array di dimensione num
uint num = espressione;
type * var = new type[num];
...
delete[] var; //per eliminare un array si usa delete con le parentesi quadre

//All'interno delle parentesi tonde si possono inserire valori di default potenziali
ObjectType* var = new ObjectType();
...
delete var; // per eliminare un oggetto si usa questa sintassi
```

Codice 1.11: Operatori `new` e `delete`

> L'uso degli operatori `new() / delete()` non va combinato con chiamate a funzioni come `malloc() / realloc() / free()`.

1.5.1 Le struct

Le **struct** in C++ hanno una doppia anima. Benché abbiano una natura molto simile alle struct definite in C, le struct sono in definitiva delle classi con accesso pubblico agli attributi e alle funzioni membro. Nella sua nozione primitiva una **struct** è una sequenza di elementi, chiamati **membri** di tipo arbitrario. Il Codice 1.12 mostra la sintassi per definire una struct.

```
struct Studente{
  string Nome;
  string Cognome;
  string Matricola;
  uint id;
};
```

Codice 1.12: Dichiarazione di una struct in C++

Per accedere ad un campo di una struttura C++ è possibile usare due strategie:

```
1 PointerVarName -> campo;
```

Codice 1.13: Accesso tramite puntatore

```
1 StructNomeVar.campo;
```

Codice 1.14: Accesso diretto

Osservazione

L'operatore **.** (dot) è applicato all'oggetto mentre l'operatore **->** (arrow) è applicato a un puntatore all oggetto (come si può notare dall'esempio precedente l'operatore freccia è `syntactic sugar` per l'operatore punto)
Scrivere **foo->bar** è equivalente a **(*foo).bar**

Un oggetto di tipo **struct** memorizza i membri secondo l'ordine col quale sono stati definiti. Ad esempio, si consideri la seguente **struct** per delle sezioni scolastiche:

```
struct classi{
  char sezione; //[A...Z]
  int numero;
  char aula;
}
```

Come già detto, i membri vengono allocati in memoria in ordine di dichiarazione, quindi l'indirizzo di `sezione` deve essere inferiore all'indirizzo della variabile `numero`. Tuttavia, la dimensione di un oggetto di una struct non è necessariamente la somma delle dimensioni dei suoi membri. Questo perché molte macchine richiedono che gli oggetti di certi tipi siano allocati su confini dipendenti dall'architettura. Ad esempio, gli interi sono spesso allocati su confini di parola. Su tali macchine, si dice che gli oggetti devono essere **allineati**. Questo porta a "buchi" nelle strutture. È possibile ridurre al minimo lo spazio sprecato **ordinando i membri in base alle dimensioni** (prima il membro più grande).

1.5.2 Enumerazioni

Quando si ha bisogno di definire un elenco di cose, è possibile definire una codifica per rappresentare questa lista. Il linguaggio offre per questo motivo il tipo di dati `enumeration`. In C++ esistono due tipologie di enumerazioni:

Enum classiche

Le **enum classiche** ereditate dal linguaggio C per le quali i nomi degli enumeratori sono nello stesso ambito dell'enum e i loro valori si convertono implicitamente in numeri interi. Per definire una enumerazione in C++ si usa la keyword `enum` come mostrato nel Listato seguente:

```
1 enum name_of_enum{
2   Element1,
3   ...,
4   ElementN,
5 };
```

Codice 1.15: Sintassi delle enumerazioni

Si consideri la seguente porzione di codice che definisce una enumerazione chiamata `ice_cream`:

```
enum ice_cream
{
  vanilla,
  chocolate,
  butterscotch,
  strawberry,
  mango,
  oreo
};
```

Ciascuno di questi elementi verrà codificato con un intero da 0 a 5. I valori di default possono essere modificati durante la dichiarazione dell'enumerazione.

Le classi enum

Per superare i limiti delle enumerazioni classiche, C++11 ha introdotto le **classi enum** (chiamate ache **enumerazioni con ambito**), che rendono le enumerazioni sia *fortemente tipizzate* e con uno *scope* ben definito. Le classi enum sono quindi delle enumerazioni per le quali i nomi degli enumeratori sono *locali* e i loro valori non si convertono implicitamente ad altri tipi.

```
// Dichiarazione
class enum NomeEnum
{
  Valore1,
  Valore2,
  Valore3
};
// inizializzazione
EnumName ObjectName = EnumName::Valore;
```

Codice 1.16: Dichiarazione di una classe enum

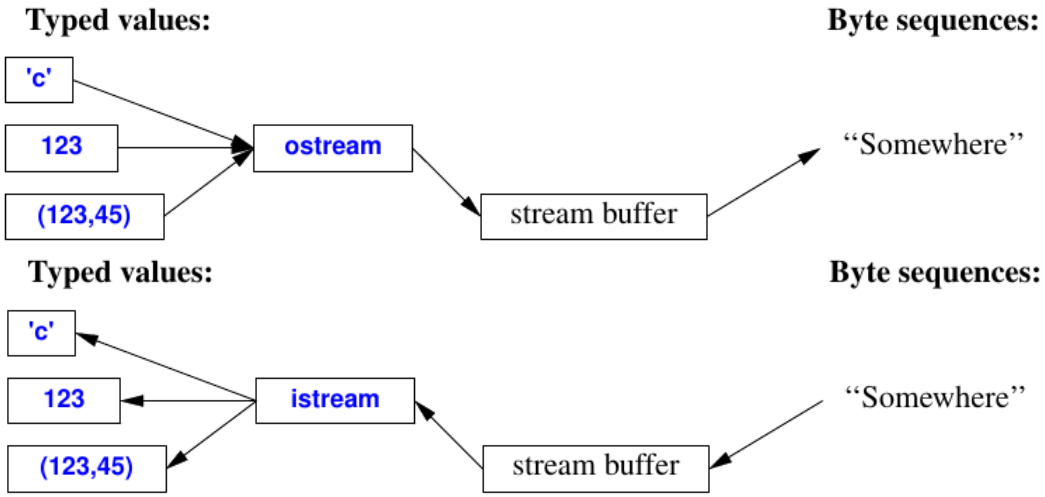
Ad esempio:

```
enum class Colore{Rosso, Verde, Blu};
Colore col = Colore::Rosso;
```

```
#include <iostream>
using namespace std;
int main() {
  enum class Color { Red, Green,  Blue };
  enum class Color2 { Red,Black,White };
  enum class People { Good,Bad };

  // An enum value can now be used to create variables
  int Green = 10;
  // Instantiating the Enum Class
  Color x = Color::Green;
  // Comparison now is completely type-safe
  if (x == Color::Red)
    cout << "It's Red\n";
  else
    cout << "It's not Red\n";

  People p = People::Good;

  if (p == People::Bad)
    cout << "Bad people\n";
  else
    cout << "Good people\n";

  // if(x == p) // gives an error
  //      cout<<"red is equal to good";
  // cout<< x;  // won't work as there is no implicit conversion to int
  cout << int(x) << endl;
  return 0;
}
```

La libreria I/O stream fornisce un I/O bufferato formattato e non formattato di valori testuali e numerici. Le definizioni per le strutture dei flussi di I/O si trovano in:

- `<ostream>`: converte oggetti in una stringa di caratteri;
- `<istream>`: converte una stringa di caratteri in oggetti.



Gli operatori principali della libreria `iostream` sono:

- L'operatore "`<<`", detto **put_to** che vanno applicati ad oggetti di tipo **ostream**;
- L'operatore "`>>`", detto **get_from** che vanno applicati ad oggetti di tipo **istream**.

Esistono quattro tipi di stream standard:

Standard stream	
`cout`	Standard output
`cin`	Standard input
`cerr`	Standard error output
`clog`	Standard error output

Tabella 1.2: Standard streams

Esempio

Si consideri la struct `Studente` definita come in 1.12. In C++ è possibile ridefinire gli operatori `put_to` e `get_from`:

```
1  ostream& operator << (ostream& os, const Studente& st)
2  {
3    os << "Nome: " <<st.Nome << "; Cognome: " << st.Cognome<<"; Matricola:  "<<st.matricola<<"; ID :"
         << st.Id << endl;
4    return os;
5  }
6
7  istream& operator >>(istream& instr, Studente& stu)
8  {
9    instr>>stu.Id >> stu.Nome>>stu.Cognome>>stu.Matricola;
10   return instr;
11 }
```

Codice 1.17: Ridefinizione degli operatori `put_to` e `get_from`

1.7 La libreria string

La libreria `string` è molto simile alla libreria `String` di Java. Un oggetto di tipo `string` non va considerato come un array di char come nel linguaggio C. Per dichiarare un oggetto di tipo `string` è possibile usare la sintassi seguente, previo aver incluso la libreria `<string>`:

```
#include <string>
int main()
{
  string Nome = "Alan";
  string Nome("Alan");
  string Nome = {"Alan"};
}
```

Codice 1.18: La libreria string

Gli oggetti string sono **confrontabili lessicograficamente**:

```
1  string Stringa1 = "Alan Turing";
2  string Stringa2 = "Kurt Godel";
3  cout << "(String1 <  String2): " << (Stringa1 < Stringa2) << endl;
```

Output:

```
Alan Turing - Kurt Godel
(String1 <  String2): 1
```

1.7.1 Concatenazione di stringhe

L'operatore `+` può essere usato tra stringhe per metterle insieme ed ottenere una nuova stringa detta **concatenazione**:

```
1  string firstName = "John ";
2  string lastName = "Doe";
3  string fullName = firstName + lastName;
4  cout << fullName;
```

Output:

```
John Doe
```

La funzione append

Una stringa in C++ è un oggetto. In quanto tale può sfruttare numerose funzione che effettuano certe operazioni sulle stringhe. Per esempio, per concatenare due stringhe è possibile anche utilizzare la funzione `append()`:

```
1  string firstName = "John ";
2  string lastName = "Doe";
3  string fullName = firstName.append(lastName);
4  cout << fullName;
```

1.8 Generazione pseudo-casuale di numeri

I numeri casuali sono essenziali per molte applicazioni, come simulazioni, giochi, algoritmi basati sul campionamento, crittografia e test. Nella libreria `<random>`, vengono definite le strutture per generare numeri (pseudo)casuali. Questi numeri casuali sono sequenze di valori prodotti secondo formule matematiche, piuttosto che numeri indovinabili ("veramente casuali") che potrebbero essere ottenuti da un processo fisico, come il decadimento radioattivo o la radiazione solare. Se l'implementazione ha un dispositivo veramente casuale, sarà rappresentato come un `random_device`. La libreria fornisce quattro entità:

- Un *generatore di numeri casuali uniformi*: un oggetto che restituisce valori interi senza segno in modo tale che ogni valore nell'intervallo dei possibili risultati abbia la stessa probabilità di essere restituito.

- Un *motore di numeri casuali*: un generatore di numeri casuali uniformi che può essere creato con uno stato predefinito $E\{\}$ o con uno stato determinato da un **seme** $E\{s\}$.
- Un *adattatore di motori di numeri casuali*: un motore di numeri casuali che prende i valori prodotti da un altro motore di numeri casuali e applica un algoritmo a tali valori per fornire una sequenza di valori con diverse proprietà di casualità.
- Una *distribuzione di numeri casuali*: un oggetto che restituisce valori che sono distribuiti secondo una funzione di densità di probabilità matematica associata $p(z)$ o secondo una funzione di probabilità discreta associata $P(zi)$.

```cpp
#include <iostream>
#include <random>
using namespace std;
int main() {
  // Engine per la generazione di numeri pseudo-casuali
  // Si sfrutta random_device{}() per estrapolare un intero (la classe implementa il Seed Sequence)
  default_random_engine gen(random_device{}());
  uniform_int_distribution<type> dist(i,j);
  for(uint i = 0; i < 15; i++)
    cout << dist(gen);
  cout << endl;
}
```

Codice 1.19: Generazione pseudo-casuale di numeri

1.9 GESTIONE DELLE ECCEZIONI

La nozione di **eccezione** è fornita per aiutare a ottenere informazioni dal punto in cui viene rilevato un errore al punto in cui può essere gestito. Una funzione che non riesce ad affrontare un problema lancia un'eccezione, sperando che il suo chiamante (diretto o indiretto) possa gestire il problema. Una funzione che vuole gestire un tipo di problema lo indica catturando l'eccezione corrispondente:

- Un chiamante indica i tipi di fallimenti che è disposto a gestire specificando tali eccezioni in una clausola `catch` di un blocco `try`.
- Un blocco che non riesce a portare a termine il compito assegnato segnala il suo fallimento utilizzando un'espressione `throw`.

Si consideri l'esempio mostrato nel Codice 1.20.

```cpp
#include <iostream>
using namespace std;
void taskmaster()
{
  try
  {
    auto result = do_task();
    // utilizzo della variabile result
  }
  catch(Some_error){
    // gestione del problema
  }
}

int do_task()
{
  //...
  if(condition) // può eseguire il task
    return result;
  else
    throw Some_error{};
}
```

Codice 1.20: Gestione delle eccezioni in C++

A differenza del linguaggio Java dove ogni metodo rende noto quali sono le eccezioni che potrebbe sollevare, in C++ le funzioni che non sollevano alcuna eccezione utilizzano la clausola `noexcept`.

1.10 PUNTATORI A FUNZIONE

Allo stesso modo degli oggetti, il codice generato per un corpo di funzione è collocato in memoria da qualche parte, quindi ha un indirizzo. Per questo motivo è possibile avere un **puntatore a una funzione** così come possiamo avere un puntatore a un oggetto. Un puntatore di una funzione può essere utilizzato per chiamare la funzione e passare le funzioni come parametro di altre funzioni. Inoltre, è possibile effettuare la chiamata di funzioni diverse con lo stesso prototipo a patto che se ne conosca l'indirizzo.

In C la dichiarazione di un puntatore a funzione segue la seguente sintassi:

```
typedef type (*FunName) (parameters);
```

Codice 1.21: Puntatore a funzione in C

In C++ la dichiarazione è quasi uguale ma questa volta il nome del tipo è alla fine della dichiarazione. Il tipo del puntatore a funzione viene dichiarato attraverso un tipo chiamato `function` con parametro template.

```
#include <functional>
typedef std:function<type(lista parametri)> FunName;
```

Codice 1.22: Puntatore a funzione in C++

Esempio

Supponiamo di avere un algoritmo di sorting. Sia data una funzione con il prototipo seguente:

```
void sort (int *A, int n);
```

dove `*A` è il puntatore ad un vettore A ed `n` è la dimensione del vettore. Data questa funzione però si avrà sempre un algoritmo prefissato per ordinare il vettore. Qualora si vogliano avere diversi algoritmi di ordinamento si dovranno scrivere molteplici funzioni `Sort`. Dato che gli algoritmi di ordinamento sono basati sul concetto di ordinamento è possibile passare un terzo parametro che dica come eseguire il confronto.

```
bool LessThan(int x, int y){return (x<y)}
bool GreaterThan(int x, int y){return (x>y)}
bool (*Compare)(int, int) //Puntatore a funzione
void sort (int *A, int n, Compare comp)
{
  /*...*/
  if(comp(x,y))
  {
    /*...*/
  }
}
```

1.11 CLASSI

Una **classe** è un tipo definito dall'utente composto da membri. I tipi più comuni di membri sono gli *attributi*, i *metodi* e i *costruttori*. Le funzioni membro possono definire il funzionamento dell'inizializzazione, della copia, dello spostamento e della pulizia (distruzione). È possibile accedere ai membri utilizzando la notazione dot (se accesso all'oggetto) oppure tramite la freccia se si stanno utilizzando dei puntatori. Una classe definisce inoltre un *namespace* per i propri membri. Il costrutto per la definizione di una classe in C++ è simile a quello visto in Java: `class`.

```
1 class ClassName{
2 private:
3   //Attributi e funzioni privati
4 protected:
5   //Attributi e metodi visibili solo ad altre classi nella gerarchia
6 public:
7   //Attributi e metodi visibili a tutti
8 };
```

Codice 1.23: Definizione di una classe in C++

1.11.1 Specificatori di accesso

I **specificatori di accesso** sono parole chiave che definiscono i livelli di accesso ai membri di una classe. Uno dei principi fondamentali della programmazione orientata agli oggetti è l'**incapsulamento**, che consiste nel nascondere i dettagli di implementazione di un oggetto e mostrare solo le funzionalità. In C++ ci sono tre specificatori di accesso:

- `public`: i membri della classe dichiarati come pubblici sono accessibili da qualsiasi parte del programma. Questo significa che i membri pubblici possono essere acceduti da qualsiasi funzione all'interno del programma.
- `private`: i membri della classe dichiarati come privati non possono essere acceduti da fuori della classe. Solo le funzioni membro della stessa classe possono accedere ai membri privati. Anche le funzioni amiche possono accedere ai membri privati.
- `protected`: i membri della classe dichiarati come protetti sono accessibili solo dalle classi derivate, ma non da qualsiasi funzione esterna.

Di default, tutti i membri di una classe in C++ sono privati. Questo significa che se non si specifica alcun specificatore di accesso, i membri della classe sono privati di default.

> **!** È possibile accedere ai membri privati di una classe utilizzando le funzioni amiche oppure utilizzando i metodi pubblici della classe stessa.

L'incapsulamento assicura un maggiore controllo sull'accesso ai membri di una classe. Questo significa che possono essere impostate regole per la modifica dei valori degli attributi di una classe. Questo è possibile utilizzando i metodi pubblici della classe per modificare i valori degli attributi privati (metodi `getter` e `setter`). Nello sviluppo di strutture dati astratte, l'incapsulamento è molto utile per mantenere l'integrità dei dati.

1.11.2 Costruttori

Per ogni classe è possibile definire un costruttore. Un costruttore in C++ è un metodo speciale che viene chiamato automaticamente quando viene creato un oggetto di una classe. Un costruttore che viene invocato senza argomenti viene chiamato **costruttore di default**: questo serve ad inizializzare il valore degli attributi della classe ad un valore di default.

La sintassi per la definizione di un costruttore è la seguente:

```
class ClassName{
public:
  ClassName(); //Costruttore di default
};
```

```
#include <iostream>
using namespace std;
class Car {
  public:          // Access specifier
    string brand;  // Attribute
    string model;  // Attribute
    int year;      // Attribute
    Car(string x, string y, int z)    { // Constructor with parameters
      brand = x;
      model = y;
      year = z;
    }
};
```

Costruttori specifici

Oltre al costruttore di default esiste la possibilità di aggiungere **costruttori specifici** che prendono in ingresso un certo numero di parametri per inizializzare gli attributi della classe.

```
class ClassName {
  public:
    ClassName(int x, int y);
  private:
    int x, y;
};
```

Tra i costruttori specifici troviamo due tipi di costruttori molto importanti in C++:

- il **costruttore di copia** prende in ingresso un riferimento costante ad un oggetto della classe stessa. La semantica attesa per un copy constructor è quella di ottenere un nuovo oggetto *copiato* da un vecchio oggetto.

```
ClassName(const ClassName&);
```

Codice 1.24: Costruttore di copia

- il **costruttore di spostamento** prende in ingresso un riferimento di tipo `rvalue` ad un oggetto della classe stessa. Un move constructor ha il compito di effettuare uno *spostamento dei dati* da un oggetto vecchio al nuovo oggetto che si sta creando. Un move constructor non dovrebbe mai sollevare eccezioni.

```
ClassName(ClassName&&) noexcept;
```

Codice 1.25: Costruttore di spostamento

Se il programmatore non definisce un costruttore di copia, ci pensa il compilatore. In questo caso il costruttore fornito dal compilatore esegue una copia bit a bit degli attributi; in generale questo è sufficiente, ma quando una classe contiene puntatori è necessario definirlo esplicitamente onde evitare problemi di condivisione di aree di memoria.

È fortemente consigliato l'ultizzo della **rule of five**

1.11.3 Creazione di un oggetto

Un **oggetto** in C++ è un'istanza di una classe. Per creare un oggetto di una classe è necessario dichiarare una variabile di tipo `ClassName` e inizializzarla con il costruttore della classe. La creazione di un *riferimento ad un oggetto* avviene tramite la chiamata del costruttore mediante l'operatore `new`. La sintassi per la creazione di un oggetto è quindi la seguente:

```
ClassName var;  //Creazione di un oggetto
ClassName* ptr = new ClassName(); //Creazione di un oggetto con new, ptr è un puntatore all'oggetto
```

Per accedere ai membri di un oggetto si utilizza l'operatore `.` se si sta utilizzando un oggetto oppure l'operatore `->` se si sta utilizzando un puntatore.

```
#include <iostream>
using namespace std;

class Car {
  public:
  string brand, model;
  int year;
  Car(string x, string y, int z)  {
    brand = x;
    model = y;
    year = z;
  }
};

int main()
{
  MyClass myObj;  // Create an object of MyClass
  myObj.myNum = 15;
  myObj.myString = "Some text";
  cout << myObj.myNum << "\n";
  cout << myObj.myString;
  // Create Car objects and call the constructor with different values
  Car carObj1("BMW", "X5", 1999);
  Car carObj2("Ford", "Mustang", 1969);
  Car *carPointer = new Car("Toyota", "Corolla", 2005);
  // Print values
  cout << carObj1.brand << " " << carObj1.model << " " << carObj1.year << "\n";
  cout << carObj2.brand << " " << carObj2.model << " " << carObj2.year << "\n";
  cout << carPointer->brand << " " << carPointer->model << " " << carPointer->year << "\n";
  delete carPointer;  // To avoid memory leak
  return 0;
}
```

1.11.4 Distruttore

Poiché ogni oggetto ha una propria durata (*lifetime*) è necessario disporre anche di un metodo che permetta una corretta distruzione dell'oggetto stesso, **un distruttore**. Un distruttore è un metodo che non riceve parametri, non ritorna alcun tipo (neanche void) ed ha lo stesso nome della classe preceduto da una `~` (tilde):

```
1 ~ClassName();
```

Codice 1.26: Distruttore di una classe in C++

Il compito del distruttore è quello di assicurarsi della corretta deallocazione delle risorse dinamiche ed evitare problemi di memory leakage. Il compilatore genera per ogni classe un distruttore di default che chiama alla fine della lifetime di una variabile.

! Un oggetto di una classe viene distrutto quando:
- Esce dallo scope in cui è stato dichiarato;
- Viene eliminato esplicitamente con l'operatore `delete`;
- Viene eliminato automaticamente quando un oggetto è un membro di un altro oggetto che viene eliminato.

Il distruttore non va mai chiamato esplicitamente. Un oggetto dichiarato staticamente viene distrutto automaticamente alla fine del programma. Un oggetto dichiarato dinamicamente deve essere eliminato esplicitamente (con `delete`) per evitare memory leakage.

1.11.5 Definizione dei metodi

La definizione dei metodi di una classe può essere eseguita dentro la dichiarazione di classe, facendo seguire alla lista di argomenti una coppia di parentesi graffe racchiudente la sequenza di istruzioni oppure riportando nella dichiarazione di classe solo il prototipo e definendo il metodo fuori dalla dichiarazione di classe. La sintassi generica per la definizione di un metodo è la seguente:

```
1 [virtual] type ClassName::NameFun (parameters) [specifiers] [=assignment]
2 {
3   // Definizione del metodo
4 }
```

Dove:

- La keyword `virtual` serve per dichiarare metodi astratti (Vedi 1.12.5) e che potranno corrispondere a tanti metodi nella gerarchia, i quali saranno scelti in base alla modalità di accesso;
- Con **`specifiers`** si intendono le keyword opzionali:
  1. `const`: indica che la funzione non modifica la struttura dati su cui agisce. Ad esempio un metodo che restituisce la dimensione di un vettore può essere definito di tipo `const`.
  2. `override`: indica la presenza di una **reimplementazione** di un metodo ereditato da una classe base;
  3. `noexcept`: indica l'assenza di eccezioni sollevate internamente dal metodo.
- Per **`=assignment`** ci si riferisce a specifiche aggiuntive che è possibile fornire al compilatore:
  - `0` serve per specificare che la funzione è un **metodo virtuale puro** (vedi 1.12.5);
  - `default` è ammesso solo per costruttori e distruttori. Chiede al compilatore di costruire un costruttore o un distruttore di default.
  - `delete` serve ad eliminare un metodo esistente ereditato.

Overloading degli operatori

Il C++ consente di eseguire l'**overloading degli operatori**, tra cui quello per l'assegnamento, il confronto e lo spostamento:

```
1 bool operator ==(const ClassName&) const noexcept; //Comparison operator
2 bool operator !=(const ClassName&) const noexcept; //Comparison operator
3 ClassName& operator=(const ClassName&); //Copy assignment
4 ClassName& operator=(ClassName&&) noexcept; //Move assignment
```

Codice 1.27: Overload degli operatori

Dichiarazioni friend

In taluni casi è desiderabile che una funzione non membro (esterne) possa accedere direttamente ai membri (attributi e/o metodi) privati di una classe. Tipicamente questo accade quando si realizzano due o più classi, distinte tra loro, che devono cooperare per l'espletamento di un compito complessivo e si vogliono ottimizzare al massimo le prestazioni, oppure semplicemente quando ad esempio si desidera eseguire l'overloading degli operatori `ostream& operator<<(ostream& o, T& Obj)` e `istream& operator>>(istream& o, T& Obj)` per estendere le operazioni di I/O alla classe T che si vuole realizzare.

In situazioni di questo genere, una classe può dichiarare una certa funzione `friend` (amica) abilitandola ad accedere ai propri membri privati.

```
#include <iostream>
class Student
{
  private:
    int SecureNumber;
  public:
    ulong Id;
      string Matricola;
      string Cognome;
      string Nome;
    Student(ulong id, string nome, string matricola, string cognome){
      Id = id;
      Nome = nome;
      Matricola = matricola;
      Cognome = cognome;
    }
    friend ostream& operator<<(ostream& os, const Student& s);
    friend istream& operator>>(istream& is, Student& s);
};

ostream& operator<<(ostream& os, const Student& s)
{
  os << "Nome: " << s.Nome << "; Cognome: " << s.Cognome << "; Matricola: " << s.Matricola << "; ID: " <<
      s.Id << endl;
  return os;
}

istream& operator>>(istream& is, Student& s)
{
  is >> s.Id >> s.Nome >> s.Cognome >> s.Matricola;
  return is;
}

int main()
{
  Student s(1234, "Alan", "123456", "Turing");
  std::cout<<s.numberID<<std::endl; // Errore
  std::cout<< s; // Stampa i dati dello studente
  std::cin >> s; // Inserimento dati studente
  return 0;
}
```

1.11.6 La parola chiave this

La parola chiave `this` è un puntatore costante che punta all'oggetto corrente. La parola chiave `this` è disponibile in tutti i metodi di una classe e viene utilizzata per fare riferimento all'oggetto corrente. La sintassi per l'utilizzo di `this` è la seguente:

```
class ClassName{
  public:
    ClassName(int x) : x{x} {}
    void print() { cout << this->x << endl; }
  private:
    int x;
};
```

1.11.7 Le funzioni inline

Le funzioni **inline** sono funzioni che vengono sostituite al loro posto ogni volta che vengono chiamate. Questo permette di evitare l'overhead dovuto alla chiamata di una funzione. Per dichiarare una funzione **inline** è sufficiente anteporre la keyword `inline` alla dichiarazione della funzione stessa.

Le funzioni `inline` rendono efficiente e funzionale la costruzione dei tipi di dati astratti. Questo perché, in caso di funzioni molto piccole, il costo di chiamata della funzione è maggiore del costo di esecuzione della funzione stessa. Inoltre, le funzioni `inline` sono utili per la definizione di metodi di classe che vengono definiti all'interno della classe stessa.

```
1 class ClassName
2 {
3 public:
4   inline void print() { cout << "Hello World" << endl; }
5 };
```

1.12 EREDITARIETÀ TRA CLASSI

Le caratteristiche più interessanti e più potenti del C++ sono l'**ereditarietà** e il **polimorfismo**, anche detto *collegamento dinamico*.

L'ereditarietà semplifica il processo di creazione di nuove classi che condividono alcune caratteristiche di una classe esistente. Il collegamento dinamico contribuisce a rendere più generali i programmi cliente, mascherando loro le differenze esistenti all'interno di una gerarchia. Il collegamento dinamico consente a ognuna delle classi appartenenti ad un gruppo di disporre di una propria implementazione di un metodo in particolare. I programmi cliente possono applicare la funzione a un'istanza di una qualsiasi fra le classi, senza tener conto della classe specifica a cui l'istanza si riferisce. In fase di esecuzione, il sistema seleziona l'implementazione corretta del metodo in base al tipo dell'oggetto a cui si applica il metodo.

1.12.1 Le classi derivate

L'idea alla base dell'ereditarietà è quella di dire al compilatore che una nuova classe (detta **classe derivata**) è ottenuta da una preesistente (detta **classe base**) "copiando" il codice di quest'ultima nella classe derivata eventualmente sostituendone una parte qualora una qualche funzione membro venisse ridefinita. La classe derivata "eredita" tutti gli attributi e i metodi della sua classe di base. Possiamo poi distinguere la classe derivata dalla sua classe base:

- Aggiungendo metodi
- Aggiungendo attributi
- Ridefinendo metodi ereditati dalla classe base

La sintassi C++ prevede la specifica delle classi base precedute dal simbolo `:`.

```
class ClassName : [virtual] [private/protected/public] BaseClassName1,BaseClassName2,...,BaseClassNameN {
  // Attributi e funzioni della classe
};
```

Una classe di base può essere ereditata da più classi derivate e una classe derivata può, a sua volta, ereditare da più di una classe base. In questo caso si parla di **ereditarietà multipla**.

1.12.2 Ereditarietà pubblica, privata e protetta

Di default l'ereditarietà è privata: tutti i membri ereditati diventano cioè membri privati della classe derivata e non sono quindi parte della sua interfaccia. È possibile alterare questo comportamento richiedendo un'ereditarietà protetta o pubblica (è anche possibile richiedere esplicitamente l'ereditarietà privata), ma quello che bisogna sempre ricordare è che **non si può comunque allentare il grado di protezione di un membro ereditato** (i membri privati rimangono dunque privati e comunque non accessibili alla classe derivata):

- Con l'ereditarietà pubblica i membri ereditati mantengono lo stesso grado di protezione che avevano nella classe da cui si eredita (classe base immediata): i membri `public` rimangono `public` e quelli `protected` continuano ad essere *`protected`*;
- Con l'ereditarietà protetta i membri public della classe base divengono membri `protected` della classe derivata; quelli `protected` rimangono tali.

1.12.3 Ereditarietà multipla: il name clash

Il C++ consente l'**ereditarietà multipla**. In questo modo è possibile far ereditare ad una classe le caratteristiche di più classi basi. L'ereditarietà multipla, però, comporta alcune problematiche che non si presentano in caso di ereditarietà singola. Quella a cui si può pensare per prima è il caso in cui le stesse definizioni siano presenti in più classi base. Questo problema va sotto il nome di **name clash**. Prendiamo in esempio il caso di una gerarchia di contenitori di dati dove la classe `D` eredita dalle classi contenitore `B1` e `B2` definite come segue:

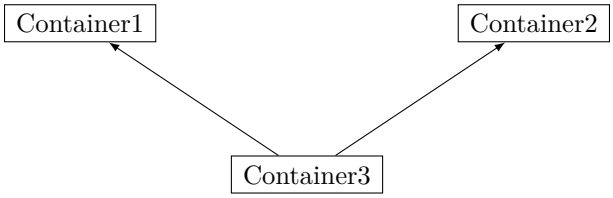


Figura 1.5: Ereditarietà multipla

```
class Base1{
  public:
  void f();
};
```

```
class Base2{
  public:
    void f();
    void f2();
};
```

```
class Derived : public Base1, public Base2{
  // Non ridefinisce f()
};

Derived d;
d.f(); // Errore: ambiguità
```

Ci si aspetterebbe di trovare in `Derived` un solo metodo `f()` però, data la gerarchia, la classe finisce per ereditare più volte gli stessi membri. In questa situazione istruzioni come mostrato nel Codice precedente risultano molto ambigue perché il compilatore non sa a quale membro si riferisce nell'assegnamento.

Si noti che l'errore viene segnalato a tempo di *esecuzione* e non al momento in cui `Derived` eredita. Il fatto che un membro sia ereditato più volte non costituisce di per sé alcun errore. Rimane ora il problema di eliminare l'ambiguità nella chiamata di `f()`. Qualora si volesse scegliere una tra le varie copie ereditate è necessario specificare il percorso utilizzando il **risolutore di scope** e indicando esplicitamente a quale implementazione del metodo ci si riferisce.

```
1 Derived d;
2 d.Base1::f(); // Chiamata a f() di Base1
3 d.Base2::f(); // Chiamata a f() di Base2
```

1.12.4 Diamond problem ed ereditarietà virtuale

Un altro problema ricorrente con l'ereditarietà multipla è la definizione di classi derivate a partire da classi base che condividono un antenato comune. Questo problema va sotto il nome di **diamond problem** dalla forma della gerarchia come mostrato in Figura 1.6. Davanti a questo problema viene in soccorso la keyword `virtual` e il concetto di **ereditarietà virtuale**.

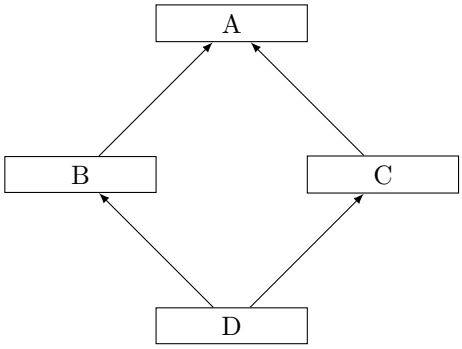


Figura 1.6: Diamond problem

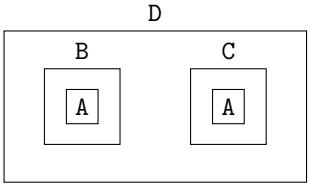


Figura 1.7: Footprint in memoria di un oggetto con ereditarietà multipla

Senza l'ereditarietà virtuale, se due classi B e C ereditano da una classe A, ed una classe D eredita da B e da C, allora D conterrà due copie dei membri della classe A: una copia tramite B e una copia tramite C. Queste copie saranno accessibili indipendentemente tramite il risolutore di scope (vedi Figura 1.7).

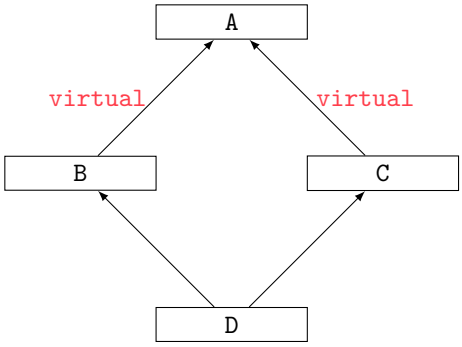


Figura 1.8: Ereditarietà virtuale

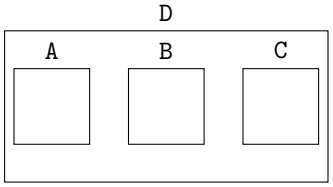


Figura 1.9: Footprint in memoria di un oggetto con ereditarietà virtuale

Al contrario, attraverso l'utilizzo della keyword `virtual` si assicura che *esista un'unica copia* nella gerarchia di quel tipo di dati. Questa funzionalità è molto utile per l'ereditarietà multipla, in quanto rende la classe base virtuale un *sottooggetto comune per la classe derivata e per tutte le classi che derivano da esse*. Questo può essere utilizzato per evitare il problema del diamante, chiarendo l'ambiguità su quale classe antenata utilizzare, poiché dal punto di vista della classe derivata (D nell'esempio precedente) la base virtuale (A) si comporta come se fosse la classe base diretta di D, non una classe derivata indirettamente attraverso una base (B o C) (vedi Figura 1.10).

L'**ereditarietà virtuale** si usa quando l'ereditarietà rappresenta la restrizione di un insieme piuttosto che la composizione di parti. Quando una classe eredita tramite la keyword `virtual` il compilatore non si limita a copiare il contenuto della classe base nella classe derivata, ma inserisce nella classe derivata **un puntatore ad una istanza della classe base**. Quando una classe eredita (per ereditarietà multipla) più volte una classe base virtuale, il compilatore inserisce solo una istanza della classe virtuale e fa si che tutti i puntatori a tale classe puntino a quell'unica istanza.

Costruttori e distruttori nelle classi derivate

Quando si costruisce un oggetto di una classe che ha più di una classe base, come mostrato in Figura 1.7, il costruttore della classe base viene chiamato nell'ordine in cui le classi base sono elencate nella definizione della classe derivata. Questo significa che il costruttore della classe base più vicina alla classe derivata viene chiamato per primo, seguito dai costruttori delle altre classi base nell'ordine in cui sono elencate. Questo perché nei vari costruttori delle classi derivate c'è sempre, implicitamente o esplicitamente, una chiamata al costruttore della classe base. Questo vale anche per i distruttori, che vengono chiamati nell'ordine inverso rispetto ai costruttori.

```
#include <iostream>
using namespace std;
class A{
  public:
    A() { cout << "base class A constructor" << endl; }
    ~A() { cout << "base class A destructor" << endl; }
};

class B : A {
  public:
    B() { cout << "derived class B constructor" << endl; }
    ~B() { cout << "derived class B destructor" << endl; }
};

class C : A {
  public:
    C() { cout << "derived class C constructor" << endl; }
    ~C() { cout << "derived class C destructor" << endl; }
};

class D : B, C {
  public:
    D() { cout << "derived class D constructor" << endl; }
    ~D() { cout << "derived class D destructor" << endl; }
};

int main(){
  D d;
}
```

Il codice del Listato 1.12.4 produce il seguente output:

```
base class A constructor
derived class B constructor
base class A constructor
derived class C constructor
derived class D constructor
derived class D destructor
derived class C destructor
base class A destructor
derived class B destructor
base class A destructor
```

Nel caso dell'ereditarietà virtuale, invece, il costruttore della classe base virtuale viene chiamato solo una volta, indipendentemente da quante classi derivate ereditano la classe base virtuale. Questo perché la classe base virtuale è condivisa da tutte le classi derivate. Lo standard stabilisce che il compito di inizializzare la classe base virtuale spetta alla **classe massimamente derivata**.

```cpp
#include <iostream>
using namespace std;

class A{
  public:
    A() { cout << "base class A constructor" << endl; }
    ~A() { cout << "base class A destructor" << endl; }
};

class B : virtual A {
  public:
    B() { cout << "derived class B constructor" << endl; }
    ~B() { cout << "derived class B destructor" << endl; }
};

class C : virtual A {
  public:
    C() { cout << "derived class C constructor" << endl; }
    ~C() { cout << "derived class C destructor" << endl; }
};

class D : B, C {
  public:
    D() { cout << "derived class D constructor" << endl; }
    ~D() { cout << "derived class D destructor" << endl; }
};

int main(){
  D d;
}
```

Il codice del Listato precedente produce il seguente output:

```
base class A constructor
derived class B constructor
derived class C constructor
derived class D constructor
derived class D destructor
derived class C destructor
derived class B destructor
base class A destructor
```

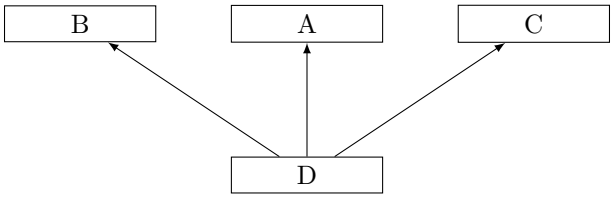


Figura 1.10: Effetto dell'ereditarietà virtuale

1.12.5 Funzioni virtuali: la keyword virtual

L'ereditarietà pone nuove regole circa la compatibilità dei tipi, in particolare se `Ptr` è un puntatore di tipo `T`, allora `Ptr` può puntare non solo a istanze di tipo `T` ma anche a istanze di classi derivate da `T` (sia tramite ereditarietà semplice che multipla). Se `Td` è una classe derivata (anche indirettamente) da `T`, istruzioni del tipo:

```
T* Ptr = nullptr;
// ...
Ptr = new Td;
```

sono assolutamente lecite e il compilatore non segnala errori o warning. Come anticpato, questo comportamento è garantito dal *dynamic binding* o *collegamento dinamico*. Ciò consente ad esempio la realizzazione di una lista per contenere tutta una serie di istanze di una gerarchia di classi, magari per poter eseguire un loop su di essa e inviare a tutti gli oggetti della lista uno stesso messaggio. Ad esempio, supponiamo di avere una gerarchia di classi che rappresentano delle forme geometriche che estendono la classe `Shape`:

```
class Shape{
  public:
    void Paint(){
      //...
    }
};
class Circle : public Shape{  /* implementazione di Paint()*/ };
class Square : public Shape{  /* implementazione di Paint()*/ };
class Triangle : public Shape{  /* implementazione di Paint()*/ };
```

Tutte le classi derivate da `Shape` devono implementare il metodo `Paint()`. Se si volesse creare una lista di oggetti di tipo `Shape` e chiamare il metodo `Paint()` di ciascun oggetto, si potrebbe scrivere il seguente codice:

```
Shape* Ptr[]= {new Circle(), new Square(), new Triangle()};
for(int i = 0; i < 3; i++)  Ptr[i]->Paint();  // Chiamata ambigua: non si sa quale metodo chiamare
```

Purtroppo il codice precedente non funziona come ci si aspetterebbe. Infatti, anche se tutte le classi della gerarchia possiedono un metodo `Paint()`, noi sappiamo solo che `Ptr` punta ad un oggetto di tipo `T` o `T`-derivato, non conoscendo l'esatto tipo una chiamata a `Ptr->Paint()` non può che essere risolta chiamando `Ptr->Shape::Paint()`. Il compilatore non può infatti rischiare di chiamare il metodo di una classe derivata, poiché questo potrebbe tentare di accedere a membri che non fanno parte dell'effettivo tipo dell'oggetto, chiamando il metodo della classe `T` al più il programma non farà la cosa giusta, ma non metterà in pericolo la sicurezza e l'affidabilità del sistema.

L'istruzione switch è dannosa

Si potrebbe pensare di risolvere il problema inserendo in ogni classe della gerarchia un campo che stia ad indicare l'effettivo tipo dell'istanza e risolvere il problema con una istruzione `switch-case`. Una soluzione di questo tipo funziona ma è macchinosa, allunga il lavoro e una dimenticanza può costare caro, e soprattutto ogni volta che si modifica la gerarchia di classi bisognerebbe modificare anche il codice che la usa.

Le funzioni virtuali

In C++ è quasi sempre possibile rinunciare alle istruzioni `switch` usate in questo modo, sostituendole con l'utilizzo delle **funzioni virtuali**. Una funzione virtuale è una funzione membro che si prevede venga ridefinita nelle classi derivate. Quando si fa riferimento a un oggetto della classe derivata utilizzando un puntatore o un riferimento alla classe base, è possibile chiamare una funzione virtuale per quell'oggetto ed eseguire la versione della classe derivata della funzione. Con le funzioni virtuali viene assicurata quindi la chiamata della funzione corretta per un oggetto, indipendentemente dall'espressione utilizzata per effettuare la chiamata di funzione. Per fare ciò bisogna dichiarare la funzione membro `virtual`.

```
class Shape{
  public:
    virtual void Paint(){ /* ...*/  }
};
class Circle : public Shape { /* implementazione di Paint() */ };
class Square : public Shape { /* implementazione di Paint() */ };
class Triangle : public Shape {   /* implementazione di Paint() */ };
Shape* Ptr[]= {new Circle(), new Square(), new Triangle()};
for(int i = 0; i < 3; i++)  Ptr[i]->Paint(); // Chiamata corretta
```

1.13.1 Le funzioni virtuali pure

Il meccanismo dell'ereditarietà e quello del polimorfismo possono essere combinati per realizzare delle classi il cui unico scopo è quello di stabilire una interfaccia comune a tutta una gerarchia di classi:

```
class TShape {
  public:
    virtual void Paint() = 0; // Funzione virtuale pura
    virtual void Erase() = 0; // Funzione virtuale pura
};
```

In C++, a differenza di Java, non esiste il concetto di interfaccia e tutto è una classe.

Si osservi l'inizializzazione a `=0` nelle dichiarazioni dei metodi `Paint()` ed `Erase()`. Questo indica che si tratta di **funzioni virtuali pure**. Una funzione virtuale pura è una funzione virtuale che non ha un'implementazione e deve essere ridefinita da una classe derivata. Una classe che contiene almeno una funzione virtuale pura è detta **classe astratta** e non può essere istanziata. Una classe derivata da una classe astratta deve implementare tutte le funzioni virtuali pure della classe base (a meno che non venga specificato come `delete`).

Lo specificatore `delete` serve a eliminare un metodo ereditato dalla classe base. Questo è utile quando si vuole impedire che una classe derivata erediti un metodo dalla classe base.

Le classi astratte possono comunque possedere anche attributi e metodi completamente definiti (costruttori e distruttore compresi) ma non possono comunque essere istanziate, servono solo per consentire la costruzione di una gerarchia di classi che rispetti una certa interfaccia.

1.13.2 Classi interfaccia in C++

L'ereditarietà multipla è un concetto semplice ma, come abbiamo visto, nella pratica, richiede una buona progettualità per essere reso funzionale. Tuttavia, esso è un costrutto potente che compare in quasi tutti i linguaggi OOP evoluti, in forma più o meno esplicita. Ad esempio il linguaggio Java prevede la possibilità di attribuire ad una classe molteplici interfacce, cioè schemi di comportamento che si estrinsecano mediante la definizione di soli metodi pubblici, senza alterare la definizione dello stato di un'istanza con la definizione di dati membro. Questo approccio semplifica la gestione di classi aggregate, rispetto alla ereditarietà multipla di C++, poiché disaccoppia lo stato delle istanze (cioè l'insieme dei dati membro) dalla definizione dei suoi comportamenti. Così facendo si risolvono a priori molte delle criticità proprie dell'ereditarietà multipla, che abbiamo citato in precedenza.

Il costrutto di interfaccia, che è assente nello standard del linguaggio C++, può tuttavia essere emulato mediante la definizione di classi astratte che includano esclusivamente metodi virtuali puri, il distruttore virtuale, un solo costruttore privo di argomenti, e siano prive di dati membro non statici. Sotto queste condizioni, la gestione di vincoli di ereditarietà multipla è più semplice, e solitamente è anche sintomo di una progettualità conforme ai principi della programmazione orientata agli oggetti.

L'alternativa consiste nel progettare la nostra gerarchia di classi di modo che i legami gerarchici in essa definiti la rendano assimilabile alla struttura di albero aciclico, piuttosto che a quella di grafo, epurando, così facendo, il nostro modello dati da vincoli di ereditarietà multipla.

```
1  class IShape{
2    public:
3      virtual void Paint() = 0;
4      virtual void Erase() = 0;
5      virtual ~IShape() = default;
6  };
7  class Circle : public IShape{
8    public:
9      void Paint() override { /* Implementazione */ }
10     void Erase() override { /* Implementazione */ }
11 };
12 class Square : public IShape{
13   public:
14     void Paint() override { /* Implementazione */ }
15     void Erase() override { /* Implementazione */ }
16 };
```

Il meccanismo dell'ereditarietà consente il riutilizzo di codice precedentemente scritto, l'idea è quella di riconoscere le proprietà di un certo insieme di valori (tipo) e di definirle realizzando una classe base (astratta) da specializzare poi caso per caso secondo le necessità. Quando riconosciamo che gli oggetti con cui si ha a che fare sono un caso particolare di una qualche classe della gerarchia, non si fa altro che specializzarne la classe più opportuna.

Esiste un altro approccio che per certi versi procede in senso opposto; anziché partire dai valori per determinarne le proprietà, si definiscono a priori le proprietà scrivendo codice che lavora su tipologie (non note) di oggetti che soddisfano tali proprietà (ad esempio l'esistenza di una relazione di ordinamento) e si riutilizza tale codice ogni qual volta si scopre che gli oggetti con cui si ha a che fare soddisfano quelle proprietà.

Quest'ultima tecnica prende il nome di programmazione generica ed il C++ la rende disponibile tramite il meccanismo dei **`template`**. Un template altro non è che codice parametrico, dove i parametri possono essere sia valori, sia nomi di tipo. Tutto sommato questa non è una grossa novità, le ordinarie funzioni sono già di per sé del codice parametrico, solo che i parametri possono essere unicamente valori di un certo tipo.

1.14.1 Classi template

La definizione di codice generico e in particolare di una classe template non è molto complicata, la prima cosa che bisogna fare è dichiarare al compilatore la nostra intenzione di scrivere un template utilizzando appunto la keyword `template`:

```
1 template <class T>
```

Codice 1.28: Dichiarazione di una classe template

Questa semplice dichiarazione (che non deve essere seguita da ";") dice al compilatore che la dichiarazione successiva utilizzerà un generico tipo `T` che sarà noto *solo quando tale codice verrà effettivamente utilizzato*, il compilatore deve quindi memorizzare quanto segue un po' come se fosse il codice di una funzione inline per poi istanziarlo nel momento in cui `T` sarà noto.

Una `template class` di per sé non è un tipo, una classe, un oggetto o qualsiasi altra entità. Non viene generato codice da un file sorgente che contiene solo definizioni di template. Per maggiori chiarimenti, riferisciti a queste Frequently Asked Questions (FAQ) sui template pubblicate sul sito della Standard C++ Foundation.

1.14.2 La keyword typename

Consideriamo il seguente esempio:

```
1 template <class T>
2 class TMyTemplate
3 {
4   private:
5     T::TId Object;
6 };
```

È chiaro dall'esempio che l'intenzione era quella di utilizzare un tipo dichiarato all'interno di `T` per istanziarlo all'interno del template. Tale codice può sembrare corretto, ma in effetti il compilatore non produrrà il risultato voluto. Il problema è che il compilatore non può sapere in anticipo se **`T::TId`** è un identificatore di tipo o un qualche membro pubblico di `T`.

Per default il compilatore assume che `TId` sia un membro pubblico del tipo T e l'unico modo per ovviare a ciò è utilizzare la keyword `typename` introdotta dallo standard:

```
1 template <class T>
2 class TMyTemplate
3 {
4   private:
5     typename T::TId Object;
6 };
```

La keyword `typename` indica al compilatore che l'identificatore che la segue deve essere trattato come un nome di tipo, e quindi nell'esempio precedente `Object` è una istanza di tale tipo.

È importante notare che `typename` non ha lo stesso effetto di utilizzare `typedef`. Se lo scopo è quello di definire un *alias* per **`T::TId`** il procedimento da seguire è il seguente:

```cpp
template <class T>
class TMyTemplate
{
  public:
    /*...*/
  private:
    typedef typename T::TId Alias;
    Alias Object;
};
```

Un altro modo corretto di utilizzare `typename` è nella dichiarazione di template:

```cpp
template <typename T>
class TMyTemplate{...};
```

Per implementare i vettori utilizzando i template si definisce la classe `Vector` come un template con un parametro di tipo generico `T` che rappresenta il tipo degli elementi dell'array.

> **!** Lo stesso creatore di C++, Bjarne Stroustrup, nel suo libro The C++ Programming Language sconsiglia fortemente l'uso di implementazione naïve dei vettori: va sempre usata l'implementazione fornita nella libreria standard.

La definizione della classe potrebbe essere la seguente:

- `vector.hpp`, che contiene la *definizione della classe template e dei suoi metodi* ed eventuali *metodi inline o template che devono essere inclusi nel file di intestazione*;
- `vector.cpp`, che contiene *l'implementazione dei metodi della classe.*

```cpp
#ifndef VECTOR_HPP
#define VECTOR_HPP

template <typename T>
class Vector {
public:
    Vector() = default;
    ~Vector();
    void push_back(const T& value);
    // Altri metodi della classe Vector
private:
    T* data;
    int size;
    int capacity;
};

#endif
```

Codice 1.29: File `vector.hpp`

```cpp
#include "vector.hpp"
template <typename T>
// NB: This leads to definition of explicitly-defaulted 'Vector<T>::Vector()'
Vector<T>::Vector() : data(nullptr), size(0), capacity(0) {}

template <typename T>
Vector<T>::~Vector() {
    delete[] data;
}

template <typename T>
void Vector<T>::push_back(const T& value) {
    if (size >= capacity) {
        // Riallocazione della memoria
    }
    data[size] = value;
    size++;
}

// Altri metodi della classe Vector
```

Codice 1.30: File `vector.cpp`

È fortemente consigliato evitare di includere all'interno degli header file (.hpp) i source code (.cpp)

Il compilatore non compilerà la classe template fino a quando questa non viene istanziata con un tipo specifico. Ad esempio, per creare un vettore di interi, si istanzia la classe template `Vector` con il tipo `int` come segue:

```
Vector<int> intVector;
```

1.14.3 Funzioni template

Grazie all'utilizzo dei template possiamo scrivere funzioni generiche che possano funzionare con tipi di dato diversi. Una funzione parametrizzata viene definita seguendo la seguente sintassi:

```
template <typename T>
T functionName(T parameter1, T parameter2, ...) {
  // code
}
```

Una volta aver dichiarato e definito la funzione `functionName`, questa potrà essere richiamata mediante la seguente sintassi:

```
functionName<dataType>(parameter1, parameter2,...);
```

Ad esempio, consideriamo una funzione parametrizzata che esegue la somma di due variabili:

```
#include <iostream>
using namespace std:

template <typename T>
T add(T num1, T num2) {
  return (num1 + num2);
}

int main() {
  int result1;
  double result2;
  // calling with int parameters
  result1 = add<int>(2, 3);
  cout << result1 << endl;

  // calling with double parameters
  result2 = add<double>(2.2, 3.3);
  cout << result2 << endl;
  return 0;
}
```

Il compilatore genera automaticamente una versione della funzione per ogni tipo di dato con cui viene chiamata.

Le funzioni template possono anche avere più di un parametro diverso. Ad esempio, consideriamo una funzione che restituisce il massimo tra due valori:

```
template <typename T1, typename T2>
T1 max(T1 num1, T2 num2) {
  return (num1 > num2) ? num1 : num2;
}
```

Per richiamare la funzione `max` con due tipi di dati diversi, si deve specificare il tipo di ritorno:

```
int result1;
double result2;
result1 = max<int, double>(2, 3.3);
cout << result1 << endl;
result2 = max<double, int>(2.2, 3);
cout << result2 << endl;
```

A partire dallo standard C++11, la specifica del linguaggio prevede la possibilità di definire **funzioni anonime** mediante l'uso di **espressioni lambda**. Le espressioni lambda vengono in genere usate per incapsulare alcune righe di codice passate ad algoritmi o funzioni asincrone. L'uso di questo costrutto non incrementa la capacità espressiva del linguaggio di per sé, ma lo arricchisce di una sintassi che consente di implementare chiusure e callback in modo molto meno prolisso.

Funzioni di callback

Una funzione di callback è una funzione passata come argomento ad un'altra funzione.

1.15.1 Sintassi di base

Gli elementi di base per la definizione di una espressione lambda sono illustrati nel diagramma mostrato in Figura 1.11.

- La **clausola di cattura** contiene una lista di variabili catturate per valore o riferimento dal contesto di definizione della espressione lambda. Questa parte del costrutto si presta a definire in modo molto snello una **chiusura funzionale**, cioè l'associazione tra una sequenza di istruzioni ed un insieme di variabili appartenenti ad un contesto esterno.
- La **lista di parametri** è un elemento opzionale del costrutto, impiegata tipicamente quando una espressione lambda è usata come callback, ad esempio nell'invocazione di uno degli algoritmi della libreria standard.
- Il **corpo delle istruzioni** è composto dalla sequenza di istruzioni che manipolano i dati catturati dal contesto esterno o passati per argomento.

Essendo una funzione anonima una espressione lambda manca di identificatore. In pratica una espressione lambda è un altro modo di definire in C++ un entità invocabile che può essere usata in maniera diretta o passata come argomento ad un'altra funzione, ma la cui applicazione è di scarsa utilità al di fuori del contesto di definizione. Per questi casi d'uso, la definizione di una espressione lambda risulta meno prolissa della definizione di una funzione o un funtore ad hoc ed ha il grande vantaggio di essere localizzata nel contesto d'uso.

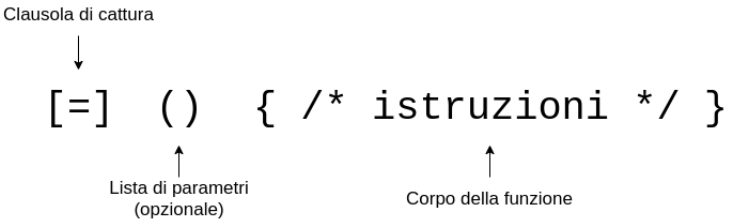


Figura 1.11: Struttura di una espressione lambda

Ecco una semplice espressione lambda passata come terzo argomento alla funzione `std::sort()`:

```cpp
#include <iostream>
#include <vector>
#include <algorithm>
int main()
{
    int num = 3;
    std::vector<int> v = {1, 2, 3, 4};
    std::transform(v.begin(), v.end(), v.begin(), [num] (const int& e) { return e * num; });
    for (auto e : v)
        std::cout << e << "\n";
    return 0;
}
```

Codice 1.31: Esempio di espressione lambda

La possibilità di catturare variabili dall'esterno è ciò che effettivamente differenzia l'uso di espressioni lambda dai tradizionali puntatori a funzione. In questa modalità d'impiego, definire una espressione lambda equivale a definire una classe dotata di un sovraccarico dell'operatore di chiamata a funzione e di dati membro in cui vengono copiati i valori o gli indirizzi delle variabili catturate, solo che è il compilatore che fa la maggior parte del lavoro per noi.

1.15.2 Cattura per valore o riferimento

La clausola di cattura può essere specializzata per istruire il compilatore sul modo corretto di effettuare l'associazione con le variabili esterne.

Le parentesi quadre possono includere una lista di identificatori preceduti separati da virgole per indicare cattura per valore. Il simbolo `=` è usato per indicare che la espressione cattura il contesto esterno nella sua interezza per valore. In alternativa è possibile specificare la cattura per riferimento apponendo il simbolo `&` come prefisso di uno o più identificatori oppure a solo per indicare che la cattura per riferimento deve essere estesa a tutte le variabili del contesto esterno che non sono esplicitamente catturate per valore.

Proprio come avviene nel caso del passaggio di parametri ad una funzione, catturare una variabile per riferimento implica che è possibile alterare il suo valore all'interno della espressione lambda. Catturare per riferimento consente inoltre di evitare di copiare i valori catturati, tuttavia nel caso di esecuzione asincrona espone a potenziali violazioni di accesso, qualora il contesto di cattura e quello di esecuzione della espressione lambda siano disgiunti.

Di seguito sono forniti alcuni esempi di cattura:

```
// cattura tutto il contesto esterno per valore ad eccezione di var
[=, &var]
// cattura tutto il contesto esterno per riferimento ad eccezione di var
[&, var]
// cattura var1 per riferimento e var2 per valore
[&var1, var2]
// cattura il contesto esterno per valore
[=]
// cattura il contesto esterno per riferimento
[&]
// nessuna cattura
[]
```

1.15.3 Valore di ritorno

Il meccanismo di inferenza dei tipi è sufficiente nella maggior parte dei casi per la corretta definizione del tipo di ritorno di una espressione lambda. Tuttavia, in caso di ambiguità o quando il corpo della funzione anonima contempla molteplici percorsi d'uscita, potrebbe essere necessario definire il tipo del valore restituito in modo esplicito mediante l'uso di un altro costrutto introdotto con lo standard C++11, detto **trailing return type**, come mostrato nel diagramma in Figura 1.12. Diversamente dal caso della definizione della firma di una funzione in cui il tipo di ritorno può essere apposto in testa o in coda, per le espressioni lambda l'uso del costrutto **trailing return type** è l'unica sintassi ammessa per la definizione del tipo restituito.

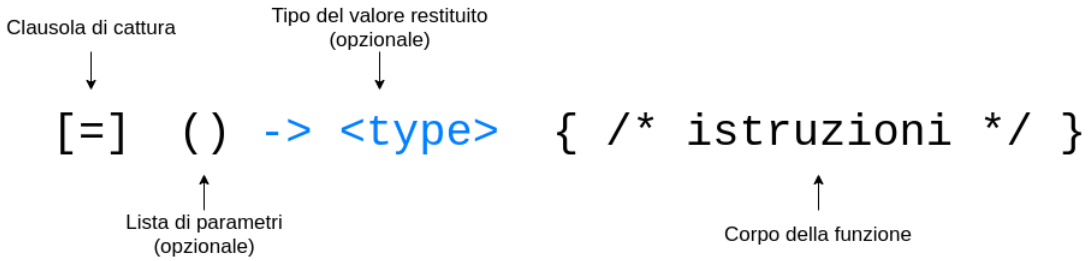


Figura 1.12: Trailing return type

Come esempio, nel Codice 1.31 è possibile aggiungere la definizione esplicita del tipo di ritorno della espressione lambda nel modo seguente:

```
[num] (const int& e) -> int { return e * num; };
```

1.15.4 Puntatori a funzione ed espressioni lambda

Le espressioni lambda possono essere utilizzate in sostituzione dei puntatori a funzione. Questo è particolarmente utile quando si stanno implementando funzioni di libreria che richiedono funzioni di callback che non sono state ancora definite o che possono variare a seconda del contesto. In questo caso, l'uso di espressioni lambda consente di scrivere codice più conciso e leggibile

rispetto all'uso di puntatori a funzione in quanto non rendono necessario dover definire una funzione separata per passarla come argomento.

Consideriamo come esempio il Codice 1.15.4. In questo esempio, l'utente può scegliere tra tre operazioni matematiche: addizione, moltiplicazione e sottrazione. L'utente inserisce due numeri e sceglie l'operazione da eseguire. L'operazione selezionata viene passata come argomento alla funzione `op()` come un'espressione lambda. La funzione `op()` prende due numeri e un puntatore a funzione come argomenti e restituisce il risultato dell'operazione applicata ai due numeri.

```
#include <functional>
#include <iostream>
typedef std::function<int(int, int)> intFunc;
int op(int, int, intFunc);

int main() {
  std::cout << "Inserisci due numeri" << std::endl;
  int a, b;
  std::cout << "a: ";
  std::cin >> a;
  std::cout << "b: ";
  std::cin >> b;
  int choice = 0;
  do {
    choice = 0;
    std::cout << "Quale operazione vuoi eseguire?\n";
    std::cout << "1. Addizione\n";
    std::cout << "2. Moltiplicazione\n";
    std::cout << "3. Sottrazione\n";
    std::cout << "4. Uscita\n";
    do {
      std::cout << "Choice: ";
      std::cin >> choice;
    } while (choice < 1 || choice > 4);
    switch (choice) {
    case (1):
      std::cout << "a+b=" << op(a, b, [](int n, int m) -> int { return n + m; })
                << std::endl;
      break;
    case (2):
      std::cout << "a*b=" << op(a, b, [](int n, int m) -> int { return n * m; })
                << std::endl;
      break;
    case (3):
      std::cout << "a - b= " << op(a, b, [](int n, int m) -> int {
        return n - m;
      }) << std::endl;
      break;
    default:
      std::cout << "Arrivederci\n";
      break;
    }
  } while (choice != 4);
  return 0;
}

int op(int a, int b, intFunc f) { return f(a, b); }
```

1.16.1 Perché esistono i Makefile?

I makefile vengono utilizzati per decidere quali parti di un programma di grandi dimensioni devono essere ricompilate. Nella stragrande maggioranza dei casi vengono compilati file C o C++. Altri linguaggi in genere hanno i propri strumenti che hanno uno scopo simile a Make. Make può essere utilizzato anche oltre la compilazione, quando è necessario eseguire una serie di istruzioni a seconda di quali file sono stati modificati.

Makefile

Un **Makefile** è un file di testo che contiene una serie di direttive per il compilatore e per il linker, che permettono di compilare e linkare un programma in modo automatico. Il comando `make` legge il Makefile e compila i file sorgenti necessari per creare l'eseguibile.

1.16.2 Struttura di un Makefile

Un Makefile è composto da una serie di **regole**. Una regola si presenta così:

```
# commento
obiettivo: dipendenza1 dipendenza2 ...
  comando1
  comando2
```

Codice 1.32: Sintassi di una regola in un Makefile

- Gli **obiettivi** sono nomi di file, separati da spazi. In genere, ce n'è solo uno per regola.
- I **comandi** sono una serie di passaggi generalmente utilizzati per creare i target.
- I **prerequisiti** sono anche i nomi dei file, separati da spazi. Questi file devono esistere prima che vengano eseguiti i comandi per la destinazione. Queste sono anche chiamate **dipendenze**.

I comandi devono essere preceduti da un tabulatore.

Una volta eseguito il comando `make`, il Makefile cerca la regola che ha come target il file specificato come argomento. Se il file non esiste o è più vecchio delle dipendenze, il Makefile esegue i comandi della regola per creare il file target.

Consideriamo il seguente Makefile:

```
grosso.txt: piccolo1.txt piccolo2.txt
  cat piccolo1.txt piccolo2.txt > grosso.txt
```

Il comando `make grosso.txt`, o semplicemente `make`, controlla che il file `grosso.txt` non esista oppure sia più vecchio dei file `piccolo1.txt` e `piccolo2.txt`. In tal caso, esegue il comando `cat piccolo1.txt piccolo2.txt > grosso.txt`. Nel caso in cui `grosso.txt` esista e sia più recente dei file `piccolo1.txt` e `piccolo2.txt`, il comando `make` non fa nulla e notifica la cosa con la seguente stringa:

```
make: 'grosso.txt' is up to date.
```

Consideriamo un esempio di Makefile più elaborato. Creiamo un file chiamato `blah.cpp` che abbia il seguente contenuto:

```
// blah.cpp
#include <iostream>
int main(){
  std::cout << "Hello, World!" << std::endl;
  return 0;
}
```

Successivamente creiamo un Makefile con il seguente contenuto:

```
blah:
  g++ -o blah blah.cpp
```

Eseguendo il comando `make`, il Makefile compila il file `blah.cpp` e crea l'eseguibile `blah`. Ma c'è un problema: se il file `blah.cpp` viene modificato e si esegue nuovamente il comando `make`, il Makefile non ricompila il file `blah.cpp` perché il file `blah` esiste già. Per risolvere questo problema, si può modificare il Makefile come segue:

```
1 blah: blah.cpp
2   g++ -o blah blah.cpp
```

Quando eseguiamo `make` nuovamente, si verificano i seguenti passaggi:

- Il primo target è selezionato perché è il primo target nel Makefile.
- Questo ha un prerequisito, `blah.cpp`.
- Make decide se eseguire il target o meno. Per fare ciò, controlla se `blah.cpp` esiste e se è più vecchio di `blah`.

Quest'ultimo passaggio è fondamentale ed è l'essenza di make . Ciò che sta tentando di fare è decidere se i prerequisiti sono cambiati dall'ultima compilazione. Cioè, se `blah.cpp` viene modificato, l'esecuzione make dovrebbe ricompilare il file. E viceversa, se `blah.cpp` non è cambiato, non dovrebbe essere ricompilato.

1.16.3 Make clean

Un'altra regola comune nei Makefile è la regola `clean`. Questa regola è utilizzata per eliminare i file temporanei e gli eseguibili generati durante la compilazione. Ecco un esempio di come potrebbe apparire la regola `clean` in un Makefile:

```
1 clean:
2   rm -f *.o blah
```

Per approfondire i Makefile e la loro sintassi, si consiglia di consultare la documentazione ufficiale di make.

Progettazione di strutture dati astratte

2.1 Abstract Data Types

La maggior parte dei linguaggi di programmazione trattano le variabili e le costanti come *istanze* di un *tipo di dato*. Un tipo di dato fornisce una descrizione delle sue istanze, che comunica al compilatore delle informazioni relative, ad esempio, alla quantità di memoria da allocare per un'istanza, alla interpretazione dei dati in memoria e alle operazioni che possono essere effettuate su quei dati.

Tipo di dato

Un **tipo di dato** è una raccolta di oggetti e una serie di *operazioni* che agistono su tali oggetti.

Alcuni linguaggi di programmazione possiedono delle risorse che consentono all'utente di estendere in modo effettivo il linguaggio, tramite l'aggiunta di tipi da lui stesso creati. Un tipo di dato definito dal programmatore viene detto **tipo di dato astratto** (ADT), per distinguerlo da un *tipo di dato fondamentale*. Il termine "astratto" è riferito alla tecnica con cui il programmatore astrae dei concetti dal vasto insieme dei dettagli della programmazione, al fine di uniformarli per creare un nuovo tipo di dato.

ADT

Un **tipo di dato astratto** è un tipo di dati organizzato in modo tale che la specifica degli oggetti e la specifica delle operazioni sugli oggetti sono distinte dalla rappresentazione degli oggetti e dall'implementazione delle operazioni.

Il linguaggio C++ consente all'utente di creare dei tipi di dati astratti, dichiarando delle classi (Vedi **??**), sulle quali è possibile operare come se fossero predefinite nel linguaggio stesso. La possibilità di estenzione del linguaggio prevista dal C++ assegna al programmatore un ruolo effettivo di progettista del linguaggio.

2.2 Le classi container

Le classi container sono delle classi fondamentali nella progettazione di librerie per strutture dati perché forniscono un'astrazione generica per la gestione dei dati. In pratica, un container è una classe astratta che fornisce un'interfaccia comune per l'accesso, la gestione e la manipolazione di un insieme di elementi. Ciò permette di creare delle librerie che siano indipendenti dal tipo di dati che si desidera gestire, fornendo al programmatore un'API generica per la manipolazione dei dati. Inoltre, i containers forniscono una gestione efficiente della memoria e la possibilità di utilizzare algoritmi generici che operano su tutti i tipi di container.

La libreria standard del C++ fornisce una serie di classi container che possono essere utilizzate per la gestione di dati. Queste classi sono definite all'interno dello spazio dei nomi `std` e sono organizzate in due categorie principali:

- **Sequence containers**: sono classi che mantengono un insieme ordinato di elementi. Questi containers permettono l'accesso sequenziale agli elementi e forniscono un'interfaccia per l'inserimento e la rimozione di elementi in posizioni specifiche.
- **Associative containers**: sono classi che mantengono un insieme di elementi ordinati secondo un criterio specifico. Questi containers permettono l'accesso agli elementi tramite una chiave e forniscono un'interfaccia per l'inserimento e la rimozione di elementi in base alla chiave.

Nella progettazione di una libreria di ADT è importante definire una gerarchia di classi container che permetta di organizzare le classi in modo modulare, in modo da facilitare la gestione e l'utilizzo delle classi stesse. Sfruttando la flessibilità del paradigma ad oggetti, la libreria standard del C++ organizza le classi container in una gerarchia di classi che permette di raggruppare le classi in base alle loro caratteristiche e funzionalità.

Le strutture dati, infatti, non rappresentano dei modelli isolati ed indipendenti ma sono strettamente correlate tra loro, in quanto alcune strutture possono essere viste come delle generalizzazioni di altre. Ad esempio, una lista può essere vista come un vettore in cui gli elementi possono essere inseriti e rimossi in qualsiasi posizione, mentre una coda può essere vista come una lista in cui gli elementi possono essere inseriti solo in coda e rimossi solo in testa.

2.2.1 La classe Container

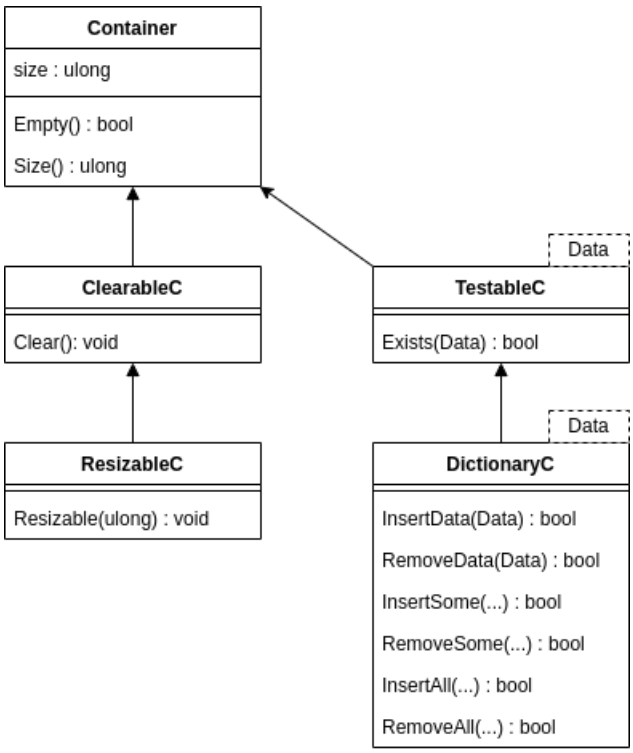


Figura 2.1: Prime classi container della libreria

In Figura 2.1 è mostrato una primo insieme di classi container della libreria che verrà implementata durante il corso. La prima classe container è la classe **Container** che rappresenta la classe base per tutte le classi container. Ciascuna struttura dati, infatti, è caratterizzata da una dimensione, ovvero il numero di elementi che essa contiene: rappresenteremo questa informazione tramite l'attributo `unsigned long size`. Inoltre, ciascuna struttura dati è caratterizzata da almeno due tipologie di test: il test di vuotezza e il test sulla dimensione della struttura. Questi test saranno implementati tramite i metodi `bool Empty()` e `unsigned long Size()`:

```
1 bool Empty() const noexcept {return (size == 0);}
2 unsigned long Size() const noexcept {return size; }
```

2.2.2 I container Clearable, Testable e Resizable

La classe `Container` viene estesa dalle classi **ClearableContainer** e **TestableContainer**. La classe `ClearableContainer` rappresenta una classe container per tutte quelle strutture dati che possono essere svuotate, ovvero che possono essere ripristinate allo stato iniziale. Questa classe estende la classe `Container` aggiungendo il metodo `void Clear()` che permette di svuotare la struttura dati. Non avendo modo di sapere come svuotare una struttura dati generica, il metodo `Clear` è definito come un metodo virtuale puro, ovvero un metodo che deve essere implementato dalle classi che estendono la classe `ClearableContainer`:

```
1 virtual void Clear() = 0;
```

La classe `TestableContainer`, invece, rappresenta una classe container per tutte quelle strutture dati che possono essere interrogate sulla presenza o meno di un elemento. Questa classe estende la classe `Container` aggiungendo il metodo `bool Exists(const Data& value)` che permette di verificare se un elemento di tipo `Data` è presente nella struttura dati. **ResizableContainer** rappresenta un'estensione della classe `Clearable` e funge da collettore per tutte quelle strutture dati che possono essere ridimensionate,

ovvero che possono modificare la propria dimensione in maniera dinamica mediante il metodo `void Resize(unsigned long new_size)` che andrà definito come metodo virtuale puro.

Osservazione

Anche se non è possibile dichiarare oggetti di una classe astratta, possiamo usare i metodi virtuali puri che questa fornisce per implementare delle funzionalità di base. Sarà poi il polimorfismo ad eseguire un dispatching dinamico per chiamare i metodi delle classi concrete che estendono la classe astratta.

Ad esempio, a questo livello di astrazione possiamo già fornire una implementazione di base per il metodo `Clear` anche senza sapere come svuotare una struttura dati generica. Questa implementazione prevede che la struttura dati venga ridimensionata a 0:

```
1 void Clear() override { Resize(0); }
```

! È sempre bene implementare le funzioni al livello più alto possibile della gerarchia di classi. In questo modo, si evita di dover reimplementare le stesse funzioni per ogni classe che estende la classe base.

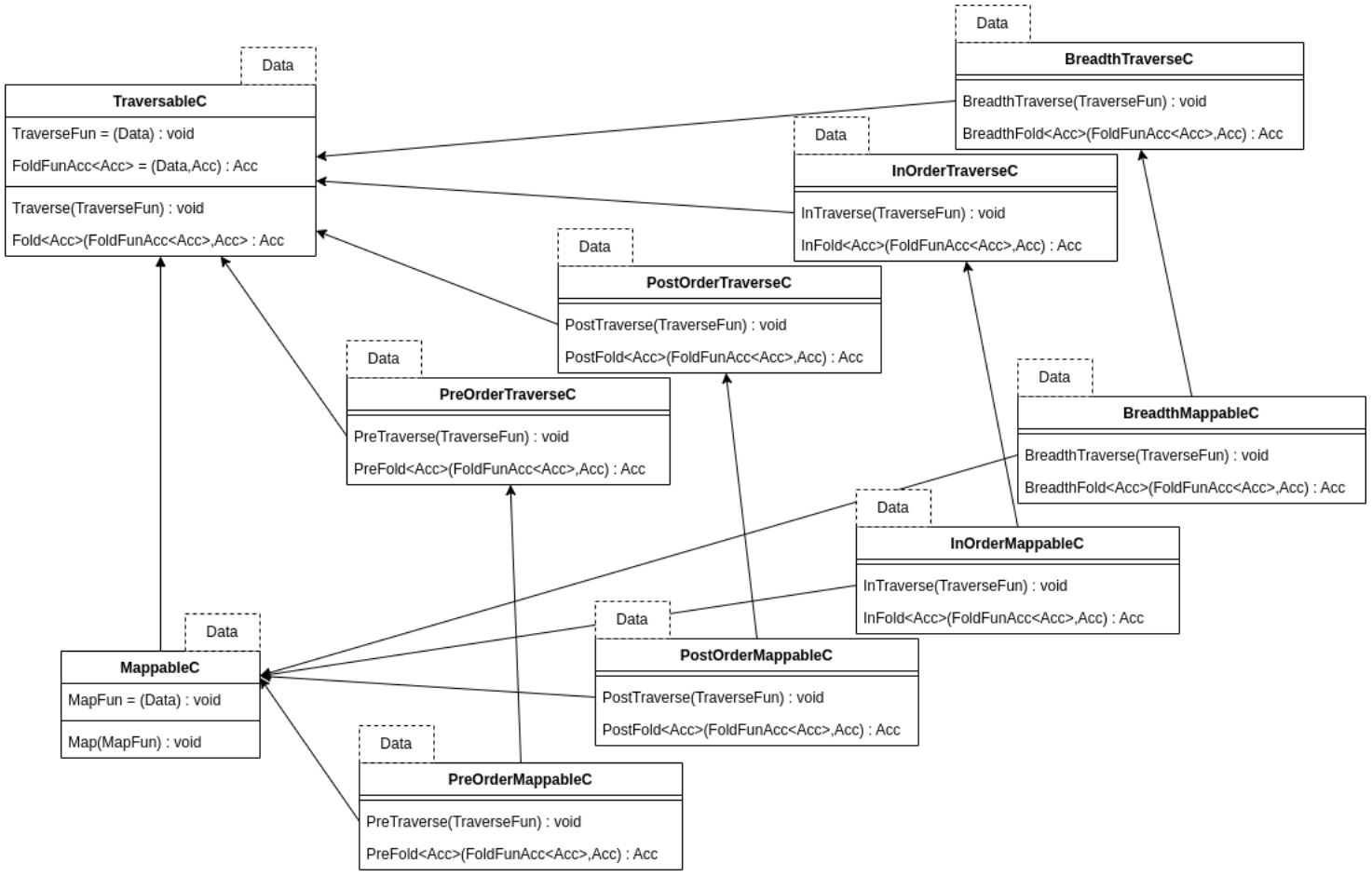


Figura 2.2: I container per l'attraversamento delle strutture dati

2.2.3 I container Traversable e Mappable

In Figura 2.2 sono mostrate le classi container per l'attraversamento delle strutture dati. La classe **TraversableContainer** rappresenta una classe container per tutte quelle strutture dati che possono essere attraversate, ovvero che permettono di accedere in lettura a tutti gli elementi della struttura dati. Una classe di tipo `TraversableContainer` estende la classe `TestableContainer` e fornisce gli attributi:

- `TraverseFun` che rappresenta un puntatore a funzione che prende in ingresso un riferimento costante ad un elemento della struttura dati e non restituisce alcun valore;

- `FoldFun` che rappresenta un puntatore a funzione che prende in ingresso un riferimento costante ad un elemento della struttura dati e un accumulatore e restituisce un accumulatore.

e i metodi:

- `void Traverse(TraverseFun)`: permette di visitare la struttura dati per mezzo di una funzione di attraversamento personalizzata;
- `Accumulator Fold(FoldFun<Accumulator>,Accumulator)`: permette di applicare una funzione di accumulazione personalizzata a tutti gli elementi della struttura dati restituendo il valore dell'accumulatore.

```
using TraverseFun = std::function<void(const Data &)>;
virtual void Traverse(TraverseFun) const = 0;

template <typename Accumulator>
using FoldFun = std::function<Accumulator(const Data &, const Accumulator &)>;
template <typename Accumulator>
inline Accumulator Fold(FoldFun<Accumulator>, const Accumulator&) const;
```

Funzione di accumulazione

Una **funzione di accumulazione** (*fold function*) è una funzione che permette di accumulare i valori di tutti gli elementi di una struttura dati.

! Le funzioni di accumulazione sono funzioni di alto livello che non permettono di modificare la struttura dati. Per questo motivo sono anche dette *visite non distruttive*. In fase di implementazione è giusto specificarle come `const` per garantire che queste non modifichino la struttura dati.

Si consideri il vettore:

1	2	3	4	5

La funzione di attraversamento `Print` permette di stampare a video tutti gli elementi del vettore. Questa può essere implementata in due modi differenti:

- Stampa dal primo all'ultimo elemento: `1 2 3 4 5`;
- Stampa dal quinto al primo elemento: `5 4 3 2 1`.

La funzione di accumulazione `Sum` permette di sommare tutti gli elementi del vettore. Tale somma viene calcolata per mezzo di un accumulatore che viene inizializzato a 0 e che viene incrementato ad ogni iterazione con il valore dell'elemento corrente. La somma dei valori del vettore è pari a 15. Il valore all'interno delle celle del vettore non viene modificato.

```
#include <iostream>
#include "Containers.h" // Inclusione della libreria dove sono definite le classi container
void Print(const int& value) {
  std::cout << value << " ";
}

void Sum(const int& value, int& accumulator) {
  accumulator += value;
}

int main() {
  /* Dichiariamo un puntatore al container di tipo TraversableContainer per sfruttare il polimorfismo:
   * il puntatore può puntare a qualsiasi oggetto che estende la classe TraversableContainer  */
  TraversableContainer<int> *v = new lasd::Vector<int>({1, 2, 3, 4, 5});
  v->Traverse(Print); // Passiamo come argomento la funzione di attraversamento
  int sum = 0;
  v->Fold<int>(Sum, &sum);  // Passiamo come argomento la funzione di accumulazione e l'accumulatore
  std::cout << sum << std::endl;
  delete v;
  return 0;
}
```

Codice 2.1: Esempio di funzione di attraversamento e di accumulazione

Se l'accesso ai dati della struttura dati è possibile anche in scrittura è possibile estendere la classe `TraversableContainer` con la classe **MappableContainer**. Tale classe definisce al suo interno un metodo `void Map(MapFunction f)` che permette di

applicare una funzione `f` a tutti gli elementi della struttura dati per mezzo di un puntatore a funzione chiamato `MapFunction` che punta ad una funzione che riceve in ingresso un riferimento ad un elemento della struttura dati.

Funzione di mappatura

Una **funzione di mappatura** (*map function*) è una funzione che permette di applicare una funzione ad ogni elemento di una struttura dati.

Si consideri il vettore:

1	2	3	4	5

La funzione di mappatura `Square` permette di elevare al quadrato tutti gli elementi del vettore. Il vettore risultante sarà:

1	4	9	16	25

```
1  #include <iostream>
2  #include "Containers.h" // Inclusione della libreria dove sono definite le classi container
3
4  int main() {
5    MappableContainer<int> *v = new lasd::Vector<int>({1, 2, 3, 4, 5});
6    std::cout << "Before: ";
7    v->Traverse(Print);
8    v->Map([](int& value)->{ value *= value; }); // Lambda expression per elevare al quadrato tutti gli
         elementi
9    std::cout << "After: ";
10   v->Traverse(Print);
11   delete v;
12   return 0;
13 }
```

Codice 2.2: Esempio di funzione di mappatura

Osservazione

Al rigo 14 del Codice 2.1 è stato utilizzato un puntatore di tipo `TraversableContainer` per sfruttare il polimorfismo. Infatti, non deve essere possibile creare un oggetto di tipo Container in quanto si tratta di una classi astratte che hanno bisogno di essere estese per poter essere utilizzate.

Grazie alle funzioni di attraversamento è possibile implementare la funzione **Exists(const Data&)** ereditata dalla classe `TestableConta` in modo molto semplice. Infatti, la funzione `Exists` può essere implementata come una funzione di attraversamento che verifica la presenza di un elemento all'interno della struttura dati. Se l'elemento è presente, la funzione restituirà `true`, altrimenti restituirà `false`.

```
1  bool Exists(const Data& value) const {
2    bool exists = false;
3    Traverse([&value, &exists](const Data& data) {
4      if (data == value) {
5        exists = true;
6      }
7    });
8    return exists;
9  }
```

La classe **DictionaryContainer** definisce al suo interno tutti quei metodi che permettono ad una struttura dati di funzionare come un dizionario.

Dizionario

Un **dizionario** è un caso particolare di insieme, in cui gli elementi di un dizionario sono coppie della forma (v, k) dove k è una chiave e v è il valore associato alla chiave k. In un dizionario non esistono coppie duplicate e deve essere possibile:

- verificare l'appartenenza di una chiave.
- inserire una nuova chiave.

- eliminare una chiave, se esistente.

Questa classe estende la classe `TestableContainer` e `MappableContainer` definendo vari metodi per l'aggiunta e la rimozione di elementi come `bool Insert(const Data& key)` e `bool Remove(const Data& key)`. Tali metodi andranno dichiarati come metodi virtuali puri in quanto non è possibile sapere come inserire o rimuovere un elemento da una struttura dati generica.

```
virtual bool Insert(const Data& key) = 0;
virtual bool Insert(Data&& key) = 0;
virtual bool Remove(const Data& key) = 0;
```

I metodi forniti dalla classe `TestableContainer` sono basati sulla specifica implementazione dei metodi `bool Insert(Data)` e `bool Remove(Data)`, definiti a livello di struttura dati concreta (Vettore, Lista, Coda, etc.). È possibile però definire già a questo livello una loro implementazione astratta per mezzo della funzione `Traverse` applicata ai container dai quali si prelevano i dati:

- `bool InsertAll(const TraversableContainer<Data>&);`
- `bool InsertAll(MappableContainer &&);`
- `bool RemoveAll(const TraversableContainer&);`
- `bool InsertSome(const TraversableContainer&);`
- `bool InsertSome(MappableContainer&&);`
- `bool RemoveSome(const TraverableContainer&);`

La differenza tra un metodo come `InsertAll` ed un metodo come `InsertSome` è che il primo *restituisce* ***`true`*** solo se inserisce tutti gli elementi della struttura dati passata come argomento, mentre il secondo inserisce `true` se inserisce almeno un elemento. Analogamente, i metodi `RemoveAll` e `RemoveSome` rimuovono tutti o alcuni elementi della struttura dati passata come argomento. L'implementazione di questi metodi sfrutta la funzione di attraversamento per prelevare gli elementi dalla struttura dati passata come argomento e inserirli o rimuoverli dalla struttura dati corrente.

```
bool InsertAll(const TraversableContainer<Data>& container) {
  bool inserted = true;
  container.Traverse([&inserted, this](const Data& data) {
    if (!Insert(data)) {
      inserted = false;
    }
  });
  return inserted;
}

bool DictionaryContainer<Data>::InsertSome(const TraversableContainer<Data>& source)
{
    bool someInserted = false;
    source.Traverse(
        [this,&someInserted](const Data & dat){
            someInserted |=Insert(dat);
    }
    );
    return someInserted;
}
```

! Nonostante una funzione di inserimento o rimozione possa essere definita come `void`, è buona norma definirla come `bool` per restituire un valore booleano che indichi se l'operazione è stata effettuata con successo. Sarà importante verificare, prima di effettuare un inserimento o una rimozione di un dato, che il dato sia presente o meno.

Estendendo la classe `TraversableContainer` per ogni tipologia di attraversamento della struttura dati possiamo definire le classi container:

- **PreOrderTraverseContainer**;
- **InOrderTraverseContainer**;
- **PostOrderTraverseContainer**

- **BreadthTraverseContainer**

che definiscono al loro interno una loro implementazione del metodo `void Traverse()`. Questi metodi permettono di attraversare la struttura dati in preordine, in ordine, in postordine e in ampiezza, rispettivamente. A questo livello di implementazione sarà inoltre possibile definire il comportamento di default per il metodo `Traverse` che permette di attraversare la struttura dati in ordine. Ad esempio:

```
// PreOrderTraversableContainer dichiara che il metodo Traverse esegue l'attraversamento in preordine
template <typename Data>
void PostOrderTraversableContainer<Data>::Traverse(TraverseFun traverseFunction) const{
    PostOrderTraverse(traverseFunction);
}
```

Se inoltre è possibile applicare una funzione di mappatura alla struttura dati, possiamo definire le classi:

- **PreOrderMapContainer**;
- **InOrderMapContainer**;
- **PostOrderMapContainer**;
- **BreadthMapContainer**

che estendono la classe `MappableContainer` e definiscono al loro interno una loro implementazione del metodo `void Map()`.

2.2.4 I container lineari

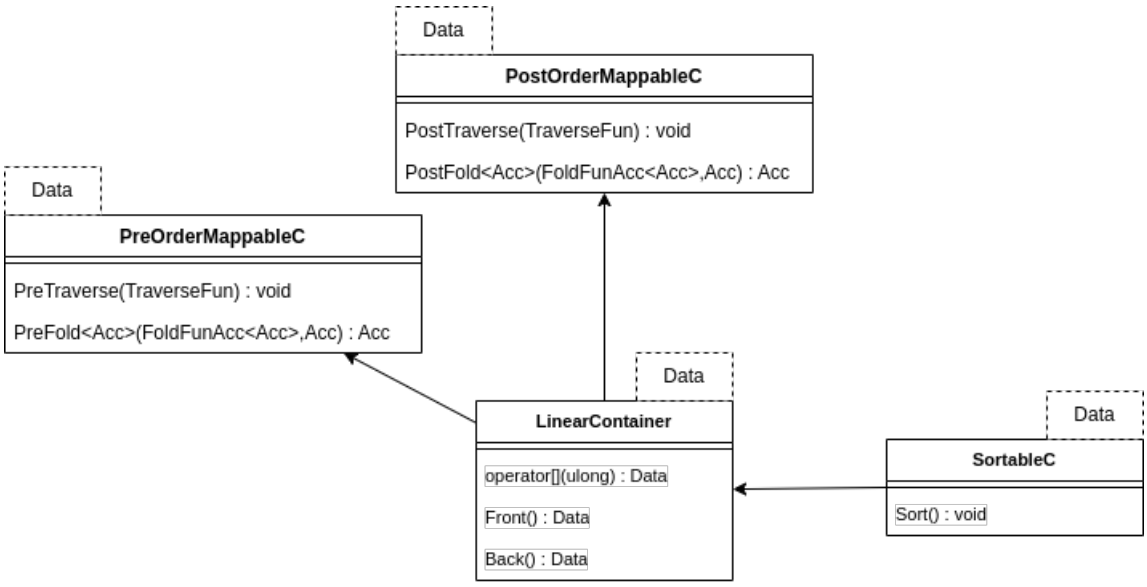


Figura 2.3: I container per le strutture dati lineari

In Figura 2.3 sono mostrate le classi container per le strutture dati lineari. La classe **LinearContainer** rappresenta una classe container per tutte quelle strutture dati che permettono di accedere agli elementi in maniera sequenziale. Questa classe estende la classe `TraversableContainer` per mezzo delle classi `PreOrderMappableContainer` e `PostOrderMappableContainer`. Un container di tipo lineare deve fornire un metodo per l'accesso in lettura e in scrittura agli elementi della struttura dati. Questo metodo è rappresentato dalla funzione `Data& operator[](unsigned long index)` che permette di accedere all'elemento in posizione `index` della struttura dati per mezzo delle convenzionali parentesi quadre. A questo livello di astrazione non è possibile fornire alcuna implementazione dell'operatore parentesi quadre in quanto non è possibile sapere come accedere ad un elemento di una struttura dati generica.

Ciò nonostante, un container di tipo lineare fornisce un metodo per la lettura dell'elemento in testa e in coda alla struttura dati. Questi metodi sono rappresentati dalle funzioni `Data& Front()` e `Data& Back()`. Tali metodi possono essere implementati sfruttando il metodo `operator[]` definito a livello di struttura dati concreta. Ad esempio, il metodo `Front` può essere implementato come segue:

```
Data & Front() const {
  if (Empty()) throw std::length_error("Container is empty");
  return operator[](0);
}
```

La classe **SortableContainer** estende infine la classe `LinearContainer` e rappresenta un'interfaccia per tutte quelle strutture dati che permettono di ordinare gli elementi in base ad un criterio specifico. Questa classe definisce al suo interno il metodo `void Sort()` che permette di ordinare gli elementi della struttura dati.

La funzione `Sort()` conterrà al suo interno una chiamata ad una funzione protetta che implementa l'algoritmo di ordinamento. Questo viene fatto per nascondere i dettagli implementativi dell'algoritmo di ordinamento all'utente che usufruisce della libreria. Ad esempio, la funzione `Sort()` può essere implementata come segue:

```
class SortableContainer {
  // SortableContainer attributes and methods
  public:
    inline void Sort() {QuickSort(0, Size() - 1);}
  protected:
    void QuickSort(unsigned long, unsigned long);
    uint Partition(unsigned long, unsigned long);
};
```

dove `QuickSort` e `Partition` sono due metodi protetti che implementano l'algoritmo di ordinamento *QuickSort*:

```
void SortableContainer::QuickSort(unsigned long i, unsigned long j) {
  if (i < j) {
    unsigned long p = Partition(i, j);
    QuickSort(i, p);
    QuickSort(p + 1, j);
  }
}

unsigned long SortableContainer::Partition(unsigned long i, unsigned long j) {
  // Partition implementation
}
```

Vettori e liste

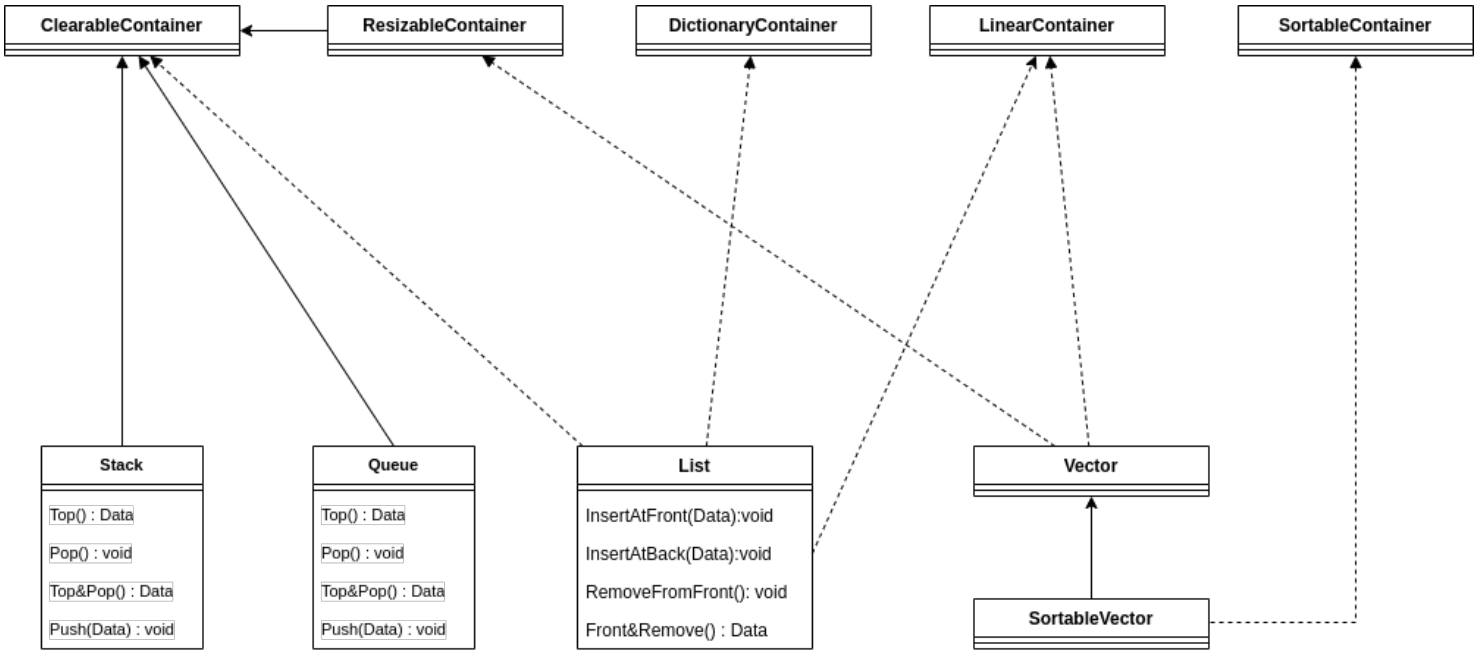


Figura 3.1: Vettori e liste

3.1 La classe Vector

Il container **Vector** è uno dei container più utilizzati in C++ e rappresenta un array dinamico. Questo container è molto simile all'array nativo di C++, ma presenta alcune funzionalità in più. In particolare, il container **Vector** permette di modificare la dimensione dell'array dinamico in modo molto semplice e veloce. Inoltre, il container **Vector** permette di accedere agli elementi dell'array dinamico in modo molto simile a come si accede agli elementi di un array nativo di C++.

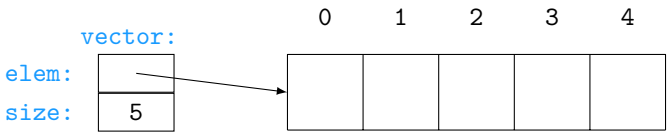


Figura 3.2: Container Vector

Come mostrato in Figura 3.2 il container **Vector** è composto da un puntatore a un array di elementi e da un intero che rappresenta la dimensione dell'array, ovvero il numero di elementi presenti nell'array.

```
template <typename Data>
class Vector : virtual public LinearContainer<Data>, virtual public ResizableContainer{
  protected:
    Data* elem; // puntatore al primo elemento dell'array
    ulong size; // dimensione dell'array
}
```

3.1.1 Costruttori

Avendo a che fare con un array dinamico, il costruttore di default del container **Vector** inizializza il puntatore a **nullptr** e la dimensione dell'array a 0. Inoltre, il container **Vector** mette a disposizione dei costruttori specifici che permettono di creare un array dinamico di una certa dimensione, di creare un array dinamico a partire da un container **TraversableContainer** e di creare un array dinamico a partire da un container **MappableContainer**.

```
Vector() = default;
Vector(const ulong) noexcept;
// Copy from TraversableContainer
Vector(const TraversableContainer<Data>&) noexcept;
// Move from MappableContainer
Vector(MappableContainer<Data>&&) noexcept;
// Copy constructor
Vector(const Vector&);
// Move constructor
Vector(Vector&&) noexcept;
```

Sarà responsabilità dei costruttori quella di richiamare l'operatore `new` per allocare la memoria necessaria per l'array dinamico e di inizializzare la dimensione dell'array dinamico.

3.1.2 Distruttori

Nel C++ moderno è diventata una good practice quella di non esporre all'utente l'utilizzo degli operatori `new` e `delete`. Questo perché l'utente potrebbe dimenticarsi di deallocare la memoria occupata dall'array dinamico, causando un **leak di memoria**. Per evitare che ciò accada, il container **Vector** deve mettere a disposizione un distruttore che si occupi di deallocare la memoria occupata dall'array dinamico.

Tale distruttore non può essere dichiarato come `default` in quanto il container **Vector** possiede un puntatore a un array di elementi.

```
~Vector()
{
  delete[] elem;
  elem = nullptr;
  size = 0;
}
```

Una volta definito un distruttore della classe `Vector` l'utente non dovrà più preoccuparsi di deallocare la memoria occupata dall'array dinamico, in quanto il distruttore si occuperà di farlo automaticamente.

3.1.3 Operatori

Il container **Vector** mette a disposizione diversi operatori per poter accedere agli elementi dell'array dinamico, per poter modificare gli elementi dell'array dinamico e per poter confrontare due array dinamici:

- Un operatore di accesso agli elementi dell'array dinamico (nelle versioni `const` e non `const`);

  ```
  Data& operator[](const ulong);
  const Data& operator[](const ulong) const;
  ```

- Un operatore di assegnamento che permette di copiare un array dinamico in un altro array dinamico (nelle versioni di copia e di spostamento);

  ```
  Vector& operator=(const Vector&);
  Vector& operator=(Vector&&) noexcept;
  ```

- Gli operatori di confronto che permettono di confrontare due array dinamici (nelle versioni di uguaglianza e di disuguaglianza).

  ```
  bool operator==(const Vector&) const;
  bool operator!=(const Vector&) const;
  ```

3.1.4 Metodi specifici

Il container **Vector**, essendo una classe che rappresenta un array dinamico, mette a disposizione una propria implementazione del metodo **Resize** e del metodo **Clear**:

```
void Resize(const ulong newSize) override
{
  Data* tmp = new Data[size];
  std::copy(elem, elem + size, tmp);
  delete[] elem;
  elem = new Data[newSize];
  std::copy(tmp, tmp + size, elem);
  delete[] tmp;
}
```

```
void Clear() override
{
  delete[] elem;
  elem = nullptr;
  size = 0;
}
```

Inoltre, per poter accedere al primo e all'ultimo elemento dell'array dinamico, il container **Vector** mette a disposizione i metodi **Front** e **Back**:

```
Data& Front() const override
{
  if(size == 0)
    throw std::out_of_range("Vector is empty");
  return elem[0];
}

Data& Back() const override
{
  if(size == 0)
    throw std::out_of_range("Vector is empty");
  return elem[size - 1];
}
```

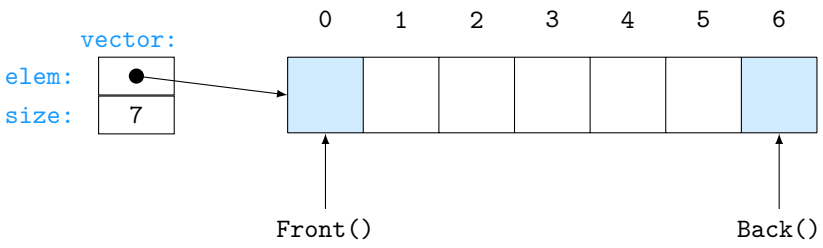


Figura 3.3: `Front()` e `Back()` in un vettore di sette elementi

3.2 La classe List

Lista

Una **lista** è una sequenza di elementi. Ogni elemento della lista è collegato al successivo e/o al precedente.

Il numero di elementi presenti un una lista è detto **lunghezza** della lista. L'inizio della lista è detto **testa** (`head`) della lista, mentre la fine della lista è detto **coda** (`tail`) della lista. Una lista si dice **vuota** quando non contiene elementi, ovvero quando il puntatore alla testa della lista è `nullptr`. Una lista si dice **circolare** quando il puntatore alla coda della lista punta alla testa della lista.

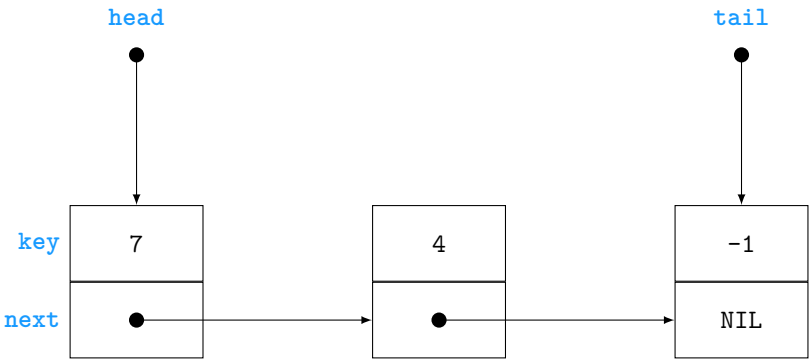


Figura 3.4: Lista concatenata

> **!** Il puntatore alla coda della lista servirà a rendere l'inserimento di un nuovo elemento in coda un'operazione di complessità costante.

Una lista è composta da **nodi**, ognuno dei quali contiene un elemento e almeno un puntatore al nodo successivo. La lista può essere implementata in due modi: come **lista concatenata** o come **lista doppiamente concatenata**. La lista concatenata è una lista in cui ogni nodo contiene un puntatore al nodo successivo, mentre la lista doppiamente concatenata è una lista in cui ogni nodo contiene un puntatore al nodo precedente e un puntatore al nodo successivo.

Come mostrato in Figura 3.4, un nodo di una lista concatenata è composto da un elemento e da un puntatore al nodo successivo. In particolare, un nodo è composto da un campo dati di tipo `Data` e da un campo puntatore di tipo `Node*`.

```
template <typename Data>
class List :  virtual public ClearableContainer,
  virtual public LinearContainer<Data>,
  virtual public DictionaryContainer<Data>
{
  protected:
    struct Node
    {
      // Node's attributes
      Data key;
      Node* next;
      // Node's methods
    }
    // Pointer to the head of the list
    Node* head;
    // Pointer to the tail of the list
    Node* tail;
    using Container::size;
  public :
    // List's methods
}
```

Nel momento in cui viene creata una lista, i puntatori alla testa e alla coda della lista vengono inizializzati a `nullptr` e la lunghezza della lista viene inizializzata a 0. Successivamente, dopo l'inserimento del primo nodo, la testa e la coda della lista punteranno entrambi al primo nodo inserito e la lunghezza della lista verrà incrementata di 1. Dopo l'inserimento di un secondo nodo, la coda della lista punterà al secondo nodo inserito e la lunghezza della lista verrà incrementata di 1. E così via.

Osservazione

Per verificare se una lista è vuota, basterà quindi verificare se il puntatore alla testa della lista è `nullptr`.

3.2.1 I costruttori

Il costruttore di default della classe **List** inizializza i puntatori alla testa e alla coda della lista a `nullptr` e la lunghezza della lista a 0.

```
List() = default;
```

Inoltre, la classe **List** mette a disposizione dei costruttori specifici che permettono di creare una lista a partire da un container `TraversableContainer` e da un container `MappableContainer` oltre che da un altro oggetto di tipo `List` (copia e spostamento).

```
// Copy from TraversableContainer
List(const TraversableContainer<Data>&);
// Move from MappableContainer
List(MappableContainer<Data>&&);
// Copy constructor
List(const List&);
// Move constructor
List(List&&) noexcept;
```

Un costruttore che effettua una copia di un oggetto di tipo `TraversableContainer` o di tipo `MappableContainer` in un oggetto di tipo `List` dovrà creare un nodo per ogni elemento del container e dovrà collegare i nodi tra loro in modo da formare una lista concatenata. Per farlo sarà innanzitutto necessario *scorrere* il container per ottenere il primo elemento e successivamente inserire i nodi nella lista per mezzo del metodo `InsertAtBack`, che come vedremo in seguito, inserisce un nodo in coda alla lista.

```
template <typename Data>
List<Data>::List(const TraversableContainer<Data> &source)
{
  source.Traverse(
    [this](const Data &data)
    {
      InsertAtBack(data);
    }
  );
}

template <typename Data>
List<Data>::List(MappableContainer<Data> &&source)
{
  source.Map(
    [this](Data &&data)
    {
      InsertAtBack(std::move(data));
    }
  );
}
```

3.2.2 Gli operatori

La classe **List**, come la classe **Vector**, mette a disposizione diversi operatori per poter accedere agli elementi della lista, per poter confrontare due liste e per poter copiare una lista in un'altra lista:

- L'**operatore parentesi quadre** (nelle versioni `const` e non `const`) permette di accedere in lettura e scrittura all'elemento di una lista in una certa posizione. Avvalendosi dell'indice dell'elemento, l'operatore scorre la lista fino a raggiungere l'elemento desiderato:

  ```
  template <typename Data>
  const Data &List<Data>::operator[](const ulong index) const
  {
    if(index < size)
    {
      Node * current = head;
      for(ulong idx = 0; idx < index; ++idx, current = current->next){}
      return current->element;
    }else
      throw std::out_of_range("Access at index " + std::to_string(index) + "out of range");
  }
  ```

- Gli **operatori di assegnamento** permettono di copiare una lista in un'altra lista (nelle versioni di copia e di spostamento). Per copiare una lista in un'altra lista, l'operatore di assegnamento deve creare un nodo per ogni nodo della lista da copiare e deve collegare i nodi tra loro in modo da formare una lista concatenata. Per farlo sarà innanzitutto necessario ripulire la lista che deve ricevere i nuovi elementi e successivamente *scorrere* la lista da copiare per ottenere il primo elemento e successivamente inserire i nodi nella lista da copiare per mezzo del metodo `InsertAtBack`:

```cpp
template <typename Data>
List<Data> &List<Data>::operator=(const List &source)
{
  if(this != &source)
  {
    Clear();
    Node * current = source.head;
    while(current != nullptr)
    {
      InsertAtBack(current->element);
      current = current->next;
    }
  }
  return *this;
}
```

Alternativamente, l'operatore di copia può essere implementato utilizzando il costruttore di copia:

```cpp
template <typename Data>
List<Data> &List<Data>::operator=(const List &source) noexcept {
  if(this != &source)
  {
    // Create a temporary list copying the source list
    List<Data> tmp(source);
    // Swap the content of the current list with the temporary list
    std::swap(head, tmp.head);
    std::swap(tail, tmp.tail);
    std::swap(size, tmp.size);
  }
  return *this;
}
```

L'operatore di spostamento, invece, può essere implementato semplicemente eseguendo uno scambio tra i puntatori alla testa e alla coda della lista da spostare e i puntatori alla testa e alla coda della lista che deve ricevere i nodi oltre che la lunghezza delle due liste:

```cpp
template <typename Data>
List<Data> &List<Data>::operator=(List &&source) noexcept
{
  std::swap(head, source.head);
  std::swap(tail, source.tail);
  std::swap(size, source.size);
  return *this;
}
```

- Gli **operatori di confronto** permettono di confrontare due liste (nelle versioni di uguaglianza e di disuguaglianza). Per confrontare due liste, l'operatore di confronto deve confrontare i nodi delle due liste uno a uno. Se i nodi delle due liste sono uguali, allora le due liste sono uguali. Se i nodi delle due liste sono diversi, allora le due liste sono diverse. Grazie all'attributo `size` possiamo evitare di confrontare due liste di lunghezza diversa che non possono essere uguali. Avendo la stessa lunghezza possiamo confrontare i nodi uno a uno sapendo che entrambe le liste finiranno nello stesso momento:

```cpp
template <typename Data>
bool List<Data>::operator==(const List &source) const{
  if(size != source.size)
    return false;
  Node * current = head;
  Node * sourceCurrent = source.head;
  while(current != nullptr)
  {
    if(current->element != sourceCurrent->element)
      return false;
    current = current->next;
    sourceCurrent = sourceCurrent->next;
  }
  return true;
}
```

3.2.3 I metodi specifici

La classe **List** essendo una classe derivata dalla classe **ClearableContainer**, mette a disposizione il metodo **Clear** che permette di eliminare tutti i nodi della lista e di deallocare la memoria occupata dai nodi della lista:

```
template <typename Data>
void List<Data>::Clear() override
{
  while(head != nullptr) RemoveFromFront();
}
```

Esempio

Per testare il funzionamento del codice che si definirà per implementare le nostre strutture dati sarà necessario definire vari **test unitari**. Tali test saranno delle funzioni che permetteranno di verificare il corretto funzionamento singolo dei metodi e degli operatori delle nostre classi.

Un test unitario deve poter essere eseguito in modo indipendente dall'input dell'utente e deve poter restituire un risultato che permetta di verificare il corretto funzionamento del metodo o dell'operatore testato in modo automatico. Ad esempio:

```
void testClear(lasd::List<Data>&list)
{
  std::cout << "Testing clear method" << std::endl;
  list.Clear();
  if(list.Empty())
    std::cout << "Test passed" << std::endl;
  else
    std::cout << "Test failed" << std::endl;
}
```

Come possiamo osservare dal codice, il test unitario `testClear` verifica se il metodo `Clear` della classe `List` funziona correttamente su una lista passata come parametro. Se il metodo `Clear` funziona correttamente, il test restituirà un messaggio di successo, altrimenti restituirà un messaggio di errore.

3.2.4 Inserimento e cancellazione di un nodo

L'inserimento di un nuovo elemento in una lista prevede i seguenti passaggi:

1. Creare un nuovo nodo e inizializzarlo con l'elemento da inserire;
2. Collegare il nuovo nodo alla lista in modo da inserirlo in una posizione specifica.

Per inserire un nodo in testa alla lista, sarà sufficiente creare un nuovo nodo facendolo puntare alla testa della lista e successivamente far puntare la testa della lista al nuovo nodo:

```
template <typename Data>
void List<Data>::InsertAtFront(const Data &data){
  Node * newNode = new Node(data);
  newNode->next = head;
  head = newNode;
  if(tail == nullptr) tail = head;
  ++size;
}
```

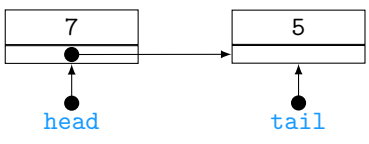


Figura 3.5: Lista prima di un inserimento in testa

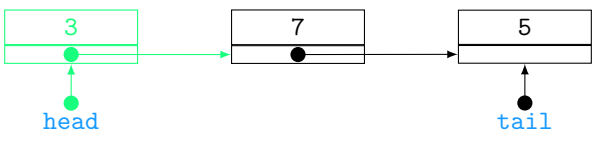


Figura 3.6: Lista dopo un inserimento in testa

Per inserire un nodo in coda alla lista, sarà sufficiente creare un nuovo nodo facendo puntare la coda della lista al nuovo nodo e successivamente aggiornare la coda della lista. Grazie al puntatore alla coda della lista, l'inserimento di un nuovo nodo in coda alla lista sarà un'operazione di complessità costante. Senza il puntatore alla coda della lista, l'inserimento di un nuovo nodo in

coda alla lista sarebbe un'operazione di complessità lineare in quanto sarebbe necessario scorrere tutta la lista fino a raggiungere l'ultimo nodo:

```
template <typename Data>
void List<Data>::InsertAtBack(const Data &data){
  Node * newNode = new Node(data);
  if(head == nullptr)
    head = newNode;
  else
    tail->next = newNode;
  tail = newNode;
  ++size;
}
```

3.2.5 La lista come un dizionario

Una lista può essere utilizzata come un dizionario, ovvero una struttura dati che permette di memorizzare coppie chiave-valore in cui la chiave è univoca. Per poter utilizzare una lista come un dizionario la classe **List** esegue l'implementazione dei metodi `Insert` e `Remove` definiti come virtuali puri nella classe astratta **DictionaryContainer**.

```
template <typename Data>
bool List<Data>::Insert(const Data &value){
  Node *current = head;
  // Search for the key in the list
  while(current != nullptr) {
    if(current->key == key)
      // If the key is found, return false
      return false;
    current = current->next;
  }
  // If the key is not found, insert the key in the list
  InsertAtBack(value);
  return true;
}
```

```
template <typename Data>
bool List<Data>::Remove(const Data &data){
  bool removed = false;
  if(head == nullptr){
    removed = false;
  }else{
    Node* current = head;
    Node* prev = head;
    while(current != nullptr && current->element != data){
      prev = current;
      current = current->next;
    }
    if(current == head)   {
      RemoveFromFront();
      removed = true;
    }else if(current==tail){
      tail = prev;
      prev->next = nullptr;
      delete current;
      removed = true;
    }else{
      prev->next = current->next;
      current->next = nullptr;
      delete current;
      removed = true;
    }
  }
  return removed;
}
```

L'eliminazione di un nodo della lista, invece, prevede i seguenti passaggi:

1. Trovare il nodo da eliminare. Se presente, passare al passo successivo, altrimenti restituire `false`. Altrimenti:
   - Se il nodo da eliminare è in testa alla lista, eliminare il nodo e far puntare la testa della lista al nodo successivo;
   - Se il nodo da eliminare è in coda alla lista, eliminare il nodo e far puntare la coda della lista al nodo precedente (se presente);
   - Se il nodo da eliminare è in mezzo alla lista:
     (a) Trovare il nodo precedente al nodo da eliminare;
     (b) Far puntare il nodo precedente al nodo successivo al nodo da eliminare;
     (c) Eliminare il nodo target.

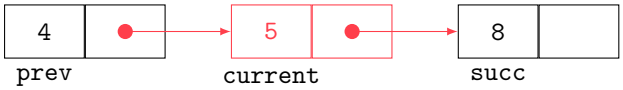


Figura 3.7: Rimozione del nodo con chiave 5 dalla lista

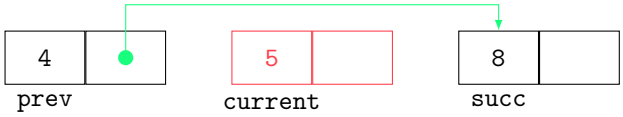


Figura 3.8: Aggiornamento dei puntatori

3.2.6 Attraversamento della lista

La classe **List** eredita dalle classi **TraversableContainer** e **MappableContainer** i metodi **Traverse** e **Map** che permettono di attraversare la lista e di applicare una funzione a tutti gli elementi della lista. Essendo un container di tipo lineare, dovremo fornire una nostra implementazione dei metodi `PreOrderTraverse`, `PostOrderTraverse`, `PreOrderMap` e `PostOrderMap`.

Sfruttando la struttura ricorsiva della lista, possiamo implementare i metodi di attraversamento e di mappatura in modo ricorsivo. In particolare, definendo dei metodi ausiliari che permettono di applicare una funzione a un nodo della lista possiamo definire i metodi di attraversamento e di mappatura in modo ricorsivo:

```
template <typename Data>
inline void List<Data>::Traverse(TraverseFun traverseFun) const {
  PreOrderTraverse(traverseFun,head);
}

template <typename Data>
inline void List<Data>::PreOrderTraverse(TraverseFun traverseFun) const{
  PreOrderTraverse(traverseFun, head);
}

template <typename Data>
inline void List<Data>::PostOrderTraverse(TraverseFun traverseFun) const{
  PostOrderTraverse(traverseFun, head);
}

template <typename Data>
void List<Data>::PreOrderTraverse(TraverseFun traverseFun, const Node *current) const{
  for (; current != nullptr; current = current->next){
    traverseFun(current->element);
  }
}

template <typename Data>
void List<Data>::PostOrderTraverse(TraverseFun traverseFun, const Node *current) const{
  if (current != nullptr){
    PostOrderTraverse(traverseFun, current->next);
    traverseFun(current->element);
  }
}
```

Per poter applicare una funzione in postordine a tutti gli elementi della lista, non avendo una lista doppiamente concatenata, è necessario ricorrere alla ricorsione. In particolare, per poter applicare una funzione in postordine a tutti gli elementi della lista, sarà necessario applicare la funzione in postordine (ovvero ponendo la chiamata ricorsiva prima dell'applicazione della funzione) al nodo successivo e successivamente applicare la funzione al nodo corrente.

Stack e code di dati

4.1 La classe Stack

Stack

Uno **stack**, o pila, è una struttura dati lineare che permette operazioni di tipo "LIFO" (Last In First Out), ovvero l'ultimo elemento inserito è il primo ad essere rimosso.

Esempio

Lo **stack di sistema** è una struttura dati che viene utilizzata per memorizzare i dati relativi alle chiamate di funzione. Ogni volta che una funzione viene chiamata, il programma crea una struttura, denominata anche **activation record** o **stack frame**, e la pone in cima allo stack di sistema.

Inizialmente, il record di attivazione per la funzione chiamata contiene soltanto un puntatore al precedente stack frame e un indirizzo di ritorno. Il puntatore del precedente stack frame punta allo stack frame della funzione che ha effettuato la chiamata, mentre l'indirizzo di ritorno contiene la posizione dell'istruzione che dovrà essere eseguita dopo il termine della funzione chiamata. Quando la funzione termina, il programma rimuove il record di attivazione dalla cima dello stack di sistema e riprende l'esecuzione dalla posizione indicata dall'indirizzo di ritorno.

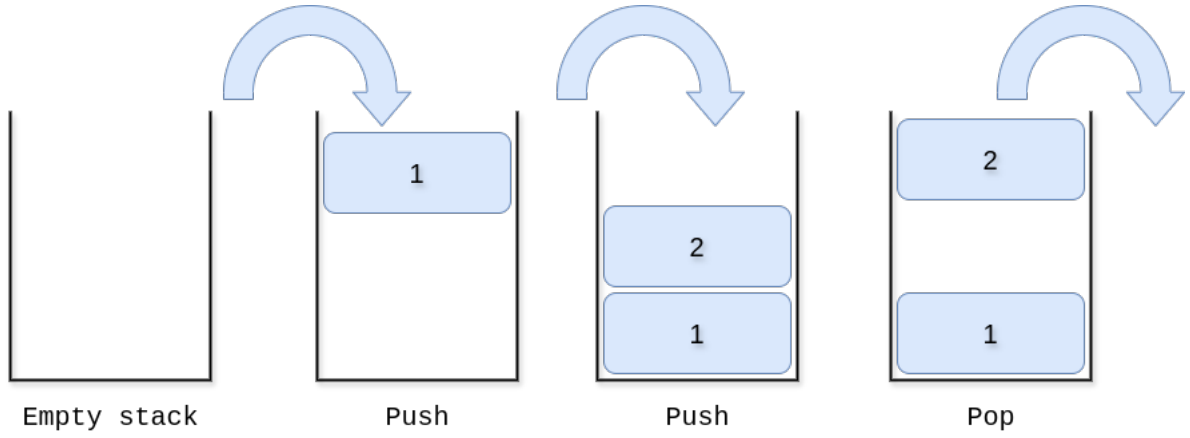


Figura 4.1: Operazioni di una pila

Un container che implemente l'ADT Stack è di tipo `ClearableContainer` e deve prevedere le seguenti operazioni come descritto nel Class Diagram 3.1:

- `void Push(T)`: inserisce un elemento di tipo `T` nello stack.
- `void Pop()`: rimuove l'elemento in cima allo stack.
- `Data& Top()`: restituisce l'elemento in cima allo stack.
- `Data TopNPop()`: restituisce l'elemento in cima allo stack e lo rimuove.

Una possibile implementazione di una interfaccia per uno stack è la seguente:

```cpp
template <typename Data>
  class Stack : virtual public ClearableContainer{

  public:
    // Destructor
    virtual ~Stack() = default;

    // Copy assignment
    Stack& operator=(const Stack& stack) = delete;

    // Move assignment
    Stack& operator=(Stack&& stack) noexcept = delete;

    // Comparison operators
    bool operator==(const Stack& stack) const noexcept = delete;
    bool operator!=(const Stack& stack) const noexcept = delete;

    // Specific member functions
    virtual const Data& Top() const = 0;
    virtual Data& Top() = 0;
    virtual void Pop() = 0;
    virtual Data TopNPop() = 0;
    virtual void Push(const Data&) = 0;
    virtual void Push(Data&&) noexcept = 0;

  };
```

Codice 4.1: Interfaccia per uno stack

4.2 La classe Queue

Queue

Una **queue**, o coda, è una struttura dati lineare che permette operazioni di tipo "FIFO" (First In First Out), ovvero il primo elemento inserito è il primo ad essere rimosso.

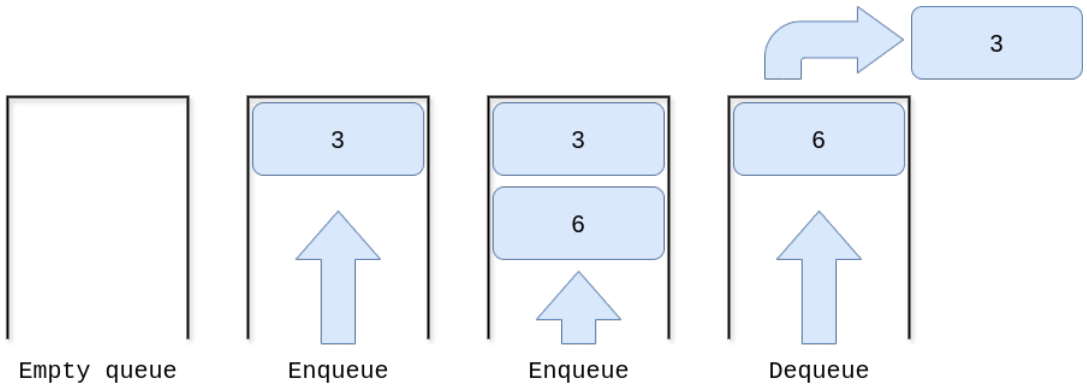


Figura 4.2: Operazioni di una coda

Un container che implemente l'ADT Queue è di tipo `ClearableContainer` e deve prevedere le seguenti operazioni come descritto nel Class Diagram 3.1:

- `void Enqueue(T)`: inserisce un elemento di tipo `T` in coda alla queue.
- `void Dequeue()`: rimuove l'elemento in testa alla queue.
- `Data& Front()`: restituisce l'elemento in testa alla queue.
- `Data FrontNDequeue()`: restituisce l'elemento in testa alla queue e lo rimuove.

Esempio

Le code sono frequentemente utilizzate nella programmazione dei sistemi operativi. Ad esempio, un sistema operativo può utilizzare una coda per gestire i processi in attesa di essere eseguiti. Quando un processo viene creato, il sistema operativo lo inserisce in coda. Quando il processore è pronto ad eseguire un nuovo processo, il sistema operativo rimuove il processo in testa alla coda e lo esegue. In questo modo, il sistema operativo garantisce che i processi vengano eseguiti nell'ordine in cui sono stati creati.

4.3 Le classi StackList e QueueList

Stack e code possono essere implementate facilmente utilizzando delle liste concatenate. Implementando l'interfaccia `Stack` e `Queue` e utilizzando le implementazioni dei metodi di `List`[1], è possibile creare le classi **StackList** e **QueueList**.

Seguendo le definizioni date per le operazioni generiche da applicare su pile, nella classe `StackLst` gli elementi dovranno essere inseriti e rimossi soltanto dalla testa della lista. Per quanto riguarda invece l'implementazione delle code mediante liste, gli accodamenti avverranno eseguendo un inserimento in coda alla lista mentre i decodamenti mediante una rimozione in testa alla lista. Sfruttando le definizioni delle operazioni definite nella ADT List, è possibile implementare le classi `StackList` e `QueueList` seguendo il seguente schema:

	Metodi esterni	Implementazione interna
StackList	`void Push(const Data&)`	`void InsertAtFront(const Data&)`
	`void Pop()`	`void RemoveFromFront()`
	`Data& Top()`	`Data & Front()`
	`Data TopNPop()`	`Data FrontNRemove()`
QueueList	`void Enqueue(const Data&)`	`void InsertAtBack(const Data&)`
	`void Dequeue()`	`void RemoveFromFront()`
	`Data& Front()`	`Data & Front()`
	`Data FrontNDequeue()`	`Data FrontNRemove()`

Tabella 4.1: Implementazioni dei metodi nelle classi `StackLst` e `QueueLst`

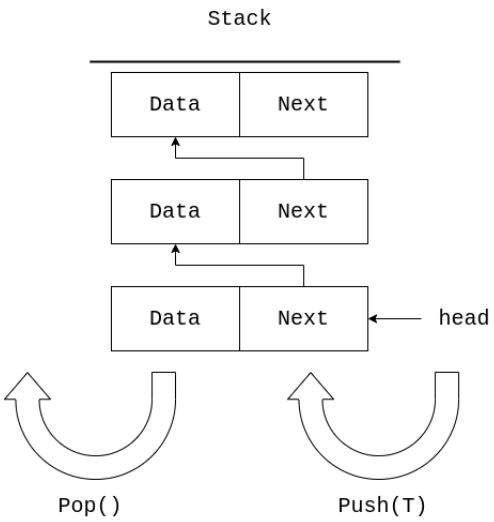


Figura 4.3: Implementazione di uno stack mediante liste

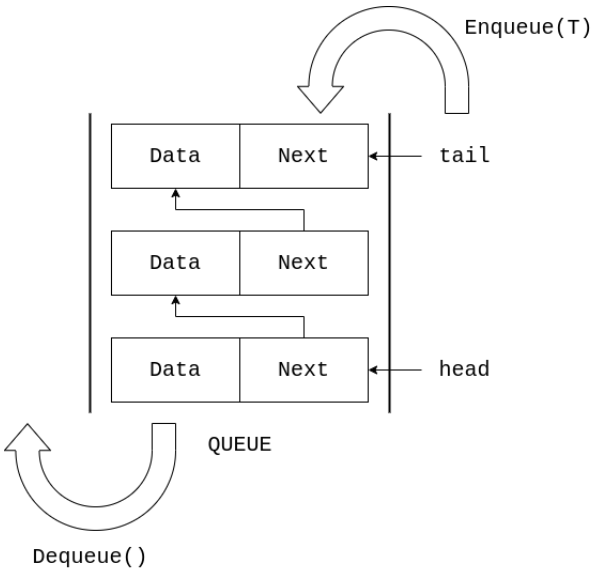


Figura 4.4: Implementazione di una coda mediante liste

Osservazione

Ricordiamo che l'inserimento in testa e in coda ad una lista concatenata ha complessità temporale $O(1)$, così come la rimozione in testa.

[1] Sarà importante, per garantire la corretta applicazione del principio di information hiding, che la classe `List` venga ereditata in modo **virtuale protetto**. In questo modo, le classi derivate avranno accesso ai metodi della classe base ma non esporranno direttamente i metodi della classe `List` al codice cliente che utilizzerà le classi derivate.

L'implementazione delle pile e delle code mediante liste si è dimostata essere molto semplice ed efficiente grazie alla struttura dati utilizzata. Tuttavia, è possibile implementare tali strutture dati astratte anche utilizzando dei vettori. In questo caso, si dovrà fare attenzione a come vengono gestite le operazioni di inserimento e rimozione degli elementi, in quanto queste operazioni potrebbero richiedere un tempo di esecuzione maggiore rispetto all'implementazione mediante liste.

4.4.1 La classe StackVec

Volendo implementare uno stack mediante un vettore, si potrebbe pensare di utilizzare un vettore di dimensione fissa e di inserire man mano gli elementi man mano fino a riempire il vettore. L'unica decisione da prendere sarebbe quella di decidere quale estremo del vettore utilizzare come cima dello stack. Una scelta possibile potrebbe essere quella di utilizzare l'indice 0 come cima dello stack, in modo tale che l'elemento in cima allo stack sia sempre in posizione `v[0]`. Tale implementazione, però, non è molto efficiente in quanto richiede di spostare tutti gli elementi del vettore ogni volta che si inserisce o si rimuove un elemento ottenendo una complessità temporale $O(n)$. Per ovviare a questo problema, dovremo distinguere quindi due variabili:

- `top`: indice della prima posizione libera del vettore.
- `size`: dimensione fisica del vettore.

In questo modo, una pila vuota avrà l'indice `top` uguale a 0. Si rende necessario così reimplementare i metodi **Empty()** e **Size()** in modo che restituiscano rispettivamente `top == 0` e `top`.

Ogni volta che si inserisce un elemento, l'indice `top` verrà incrementato di 1 e l'elemento verrà inserito in posizione `v[top]`. Ogni volta che si rimuove un elemento, l'indice `top` verrà decrementato di 1. In questo modo, l'elemento in cima allo stack sarà sempre in posizione `v[top-1]`.

Il ridimensionamento dello stack

Un problema che potrebbe sorgere utilizzando questa implementazione è quello della gestione della memoria. Infatti, se si inseriscono e si rimuovono molti elementi dallo stack, l'indice `top` potrebbe raggiungere la dimensione massima del vettore. In questo caso, non sarebbe possibile inserire nuovi elementi nello stack. Una possibile soluzione a questo problema potrebbe essere quella di raddoppiare la dimensione del vettore ogni volta che si raggiunge la dimensione massima. Inoltre, si potrebbe anche decidere di dimezzare la dimensione del vettore ogni volta che si raggiunge una certa soglia di riempimento. Sarà quindi necessario definire una soglia di riempimento, ad esempio del 25%, al di sotto della quale si dimezzerà la dimensione del vettore.

Si definiscono quindi i due metodi:

- `void Expand()`: raddoppia la dimensione del vettore.
- `void Reduce()`: dimezza la dimensione del vettore.

```
template <typename Data>
void StackVec<Data>::Expand(){
  if (top == size)
    Vector<Data>::Resize(size * 2);
}
```

```
template <typename Data>
void StackVec<Data>::Reduce(){
  if (top <= size / 4)
    Vector<Data>::Resize(size / 2);
}
```

Prima di eseguire un inserimento o una rimozione dalla pila sarà quindi necessario controllare se è necessario espandere o ridurre la dimensione del vettore. In questo modo, si garantirà che l'operazione di inserimento abbia complessità temporale $O(1)$ in media dilazionando le operazioni di ridimensionamento del vettore.

```
template <typename Data>
void StackVec<Data>::Push(const Data& data)
{
  Expand();
  Elements[top] = data;
  ++top;
}
```

```
template <typename Data>
void StackVec<Data>::Pop()
{
  if (top != 0) {
    Reduce();
    --top;
  }
  else
    throw std::length_error();
}
```

4.4.2 La classe QueueVec

L'implementazione delle code mediante array pone problemi simili a quelli visti per le pile. Ciò nonostante, applicare le stesse soluzioni viste per le pile non è possibile in quanto le code prevedono operazioni di inserimento e rimozione in due estremi diversi del vettore: se richiedessimo di avere sempre n elementi in coda, dovremmo spostare tutti gli elementi del vettore ogni volta che si inserisce o si rimuove un elemento, ottenendo una complessità temporale $O(n)$.

Infatti, supponiamo di avere un vettore di dimensione n e di fissare la testa in posizione 0. In questo caso, una eventuale operazione di decodamento comporterebbe uno spostamento di tutti gli elementi del vettore verso sinistra. Analogamente, se fissassimo la coda in posizione 0, un accodamento comporterebbe uno spostamento di tutti gli elementi del vettore verso destra. In entrambi i casi, l'operazione avrebbe complessità $O(n)$.

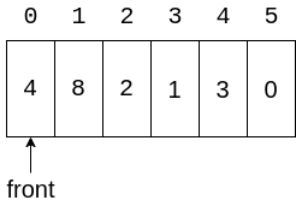


Figura 4.5: Coda di sei elementi

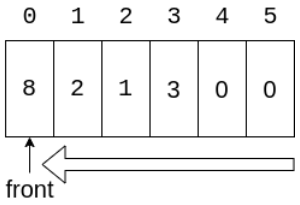


Figura 4.6: Decodamento e shift verso sinistra

Per ovviare a questo problema, si potrebbe pensare di eseguire un rilassamento sui vincoli dell'array. Infatti, se si utilizzasse un **array circolare**, sarebbe possibile accodare e decodare elementi senza dover spostare tutti gli elementi del vettore. Un array circolare è un array in cui l'elemento successivo all'ultimo elemento è il primo elemento dell'array. In questo modo, quando si raggiunge la fine dell'array, si ritorna all'inizio dell'array come mostrato in Figura 4.7.

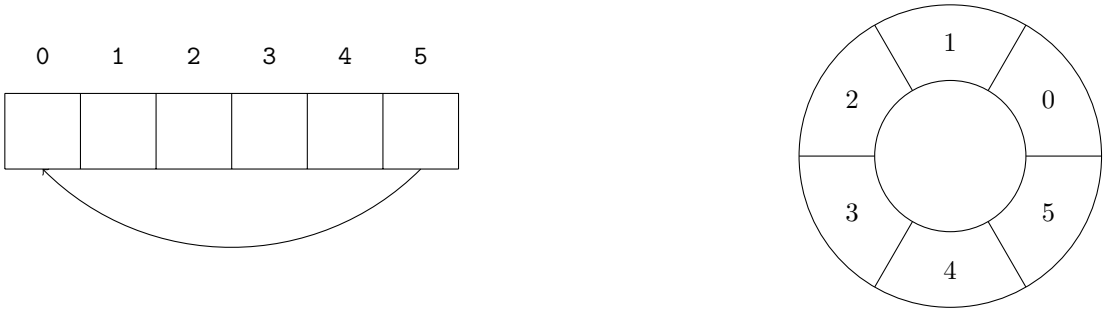


Figura 4.7: Array circolare

In un array circolare non esiste più un inizio e una fine dell'array, ma soltanto due variabili, `head` e `tail` per indicare rispettivamente il primo e l'ultimo elemento presenti nella coda. In questo modo, l'accodamento di un elemento comporterà l'incremento del puntatore `tail` e l'inserimento dell'elemento nella nuova posizione `v[tail]`. Analogamente, il decodamento di un elemento comporterà la rimozione dell'elemento in posizione `v[head]` e del successivo incremento del puntatore `head`. Ad esempio:

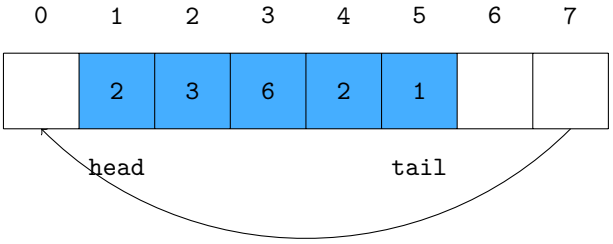


Figura 4.8: `head=2`, `tail=6`

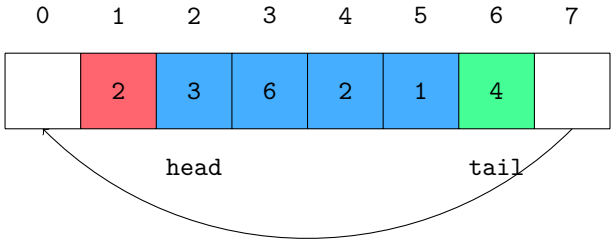


Figura 4.9: `head=3`, `tail=7`

Consideriamo di dover inserire in coda a partire dalla configurazione mostrata in Figura 4.10. In questo caso, l'elemento 4 verrà inserito in posizione 0 come mostrato in Figura 4.11.

> **!** Non è necessariamente obbligatorio rimuovere l'elemento dal vettore in quanto sarà sovrascritto successivamente da un eventuale accodamento futuro.

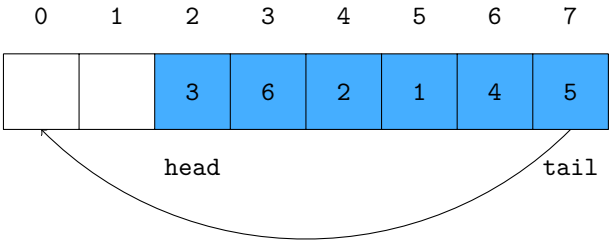


Figura 4.10: Configurazione iniziale

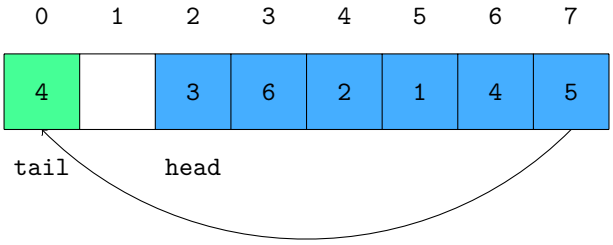


Figura 4.11: Configurazione successiva ad un inserimento in coda

L'**estrazione** (`Dequeue()`) dall'inizio della coda prevede che si estragga l'elemento di indice `head` e che si incrementi il valore di `head`. Se tale aumento porta oltre il limite dell'array, esso viene riportato a 0. Ciò si realizza usando l'operatore aritmetico modulo *size*:

```
head = (head + 1) % size;
```

Analogamente, l'**inserimento** (`Enqueue(T)`) in coda prevede che si inserisca un elemento nella cella di indice `tail` e che si incrementi il valore di `tail` per puntare alla cella successiva. Se tale aumento porta oltre il limite dell'array, esso viene riportato a 0:

```
tail = (tail + 1) % size;
```

Osserviamo che, seguendo questa implementazione, il valore di `tail` potrebbe coincidere con il valore di `head` in più di un caso. Se `head` e `tail` coincidono, potrebbe essere presente un singolo elemento nella coda oppure la coda potrebbe essere vuota.

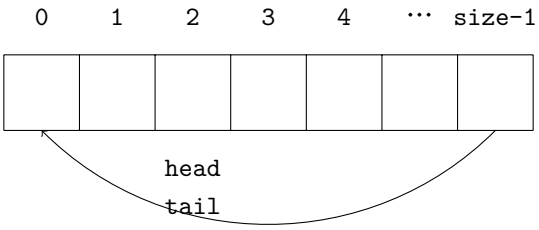


Figura 4.12: Coda vuota

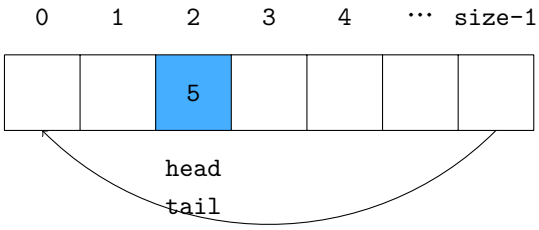


Figura 4.13: Coda con un solo elemento

È necessario quindi trovare una metodologia che ci permetta di distinguere correttamente le due situazioni. Infatti, prima di decodare un elemento, sarebbe necessario controllare se la coda sia vuota o meno, eventualmente verificando se `tail==head`. Ci sono varie possibili soluzioni per risolvere questo problema:

1. Usare un contatore per tenere traccia del numero di elementi presenti nella coda. Questo approccio prevede l'aggiunta di un campo `length` che tiene traccia del numero di elementi presenti nella coda. In questo modo, la coda sarà vuota se `length==0`.

2. Usare un array di dimensione $n + 1$ lasciando sempre una casella vuota[2]: in questo caso i due indici possono sovrapporsi solo se la coda è vuota. L'indice `head` punta al primo elemento della coda e `tail` punta alla prima posizione libera dopo l'ultimo elemento (tranne se la coda è vuota) come mostrato in Figura 4.14.

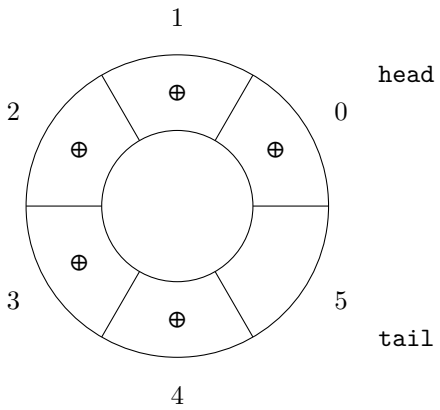


Figura 4.14: Questa coda risulta piena in quanto contiene `4 = size-1` elementi.

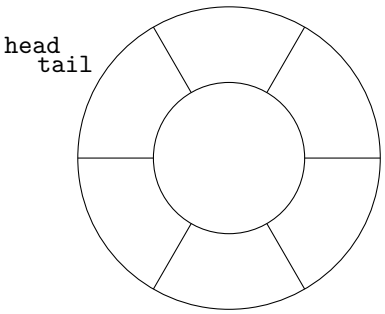


Figura 4.15: Coda vuota. Gli indici `tail` ed `head` coincidono.

Nel seguito seguiremo la seconda strategia che fa uso di una cella **sentinella**.

[2] In questo modo si potranno inserire sempre al massimo n elementi.

Per calcolare il numero di elementi in una coda implementata con un array circolare si utilizza la seguente formula:

$$\text{Numero di elementi nella coda} = \begin{cases} tail - head & head \leq tail \\ ((tail - head) + size)\%size & \text{altrimenti} \end{cases} \tag{4.1}$$

Per dimostrare la Formula 4.1 consideriamo i due casi:

1. Se `tail` è minore od uguale ad `head`, il numero di elementi nella coda è semplicemente la differenza tra i due indici.
2. Se `tail` è maggiore di `head`, poiché la coda ha "superato" il limite dell'array circolare, dovremmo aggiungere `size` all'operazione di differenza per "aggiustare" il risultato.

Per entrambi i casi, applichiamo l'operazione modulo `size` per garantire che il risultato sia nell'intervallo corretto.

L'operazione modulo `size` assicura che il risultato sia compreso tra `0` e `size-1`, che è l'indice massimo dell'array.

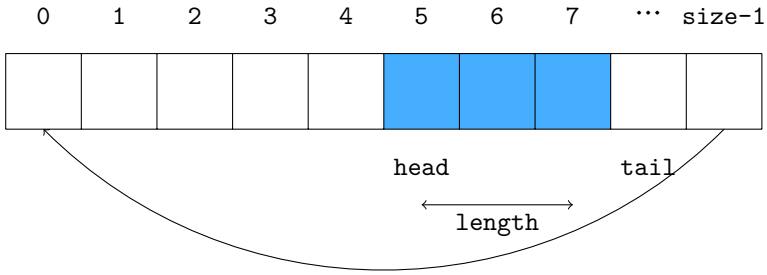


Figura 4.16: Implementazione di una coda mediante array circolare

```
template <typename Data>
bool QueueVec<Data>::Empty() const noexcept
{
  return (head == tail);
}
```

```
template <typename Data>
ulong QueueVec<Data>::Size() const noexcept
{
  if (head <= tail) return (tail - head);
  else return ((tail-head)+size)%size;
}
```

A fronte del possibile esaurimento dello spazio disponibile, si potrebbe pensare di raddoppiare la dimensione dell'array ogni volta che si raggiunge la dimensione massima. Per quanto detto in precedenza, una coda è piena quando `tail` è uguale a `head-1` oppure, equivalentemente, quando il numero di elementi presenti nella coda è pari a `size-1`. Ugualmente, si potrebbe pensare di dimezzare la dimensione dell'array ogni volta che si raggiunge una certa soglia di riempimento, ad esempio del 25%, al di sotto della quale si dimezzerà la dimensione del vettore. Si avrà quindi:

```
template <typename Data>
void QueueVec<Data>::Expand(){
  if (Size() == size - 1){
    QueueVec<Data> *tmp = new QueueVec<Data>();
    tmp.Resize(size * 2);
    SwapVectors(*tmp);
    delete tmp;
  }
}
```

```
template <typename Data>
void QueueVec<Data>::Reduce(){
  if (Size() <= size / 4){
    QueueVec<Data> *tmp = new QueueVec<Data>();
    tmp.Resize(size / 2);
    SwapVectors(*tmp);
    delete tmp;
  }
}
```

Le funzioni `Expand()` e `Reduce()` vanno chiamate prima di eseguire un inserimento o una rimozione dalla coda. In questo modo, si garantirà che l'operazione di inserimento abbia complessità temporale $O(1)$ in media dilazionando le operazioni di

ridimensionamento del vettore. Per eseguire la copia degli elementi da un vettore all'altro, si potrebbe pensare di utilizzare un metodo `SwapVectors()` che esegue una copia linearizzata della coda, riponendo la testa della coda nella prima cella del nuovo vettore e tutti gli altri elementi a seguire fino a `tail` e lasciando lo spazio libero nella parte restante del vettore come mostrato in Figura 4.18.

```
template <typename Data>
void QueueVec<Data>::SwapVectors(QueueVec<Data>& q){
  ulong index = 0;
  for (ulong i = head; i != tail; i = ((i + 1)%size)) {
    std::swap(Elements[i], source.Elements[index]);
    index++;
  }
  source.head = 0;
  source.tail = index;
  std::swap(Elements, source.Elements);
  std::swap(size, source.size);
  std::swap(head, source.head);
  std::swap(tail, source.tail);
}
```

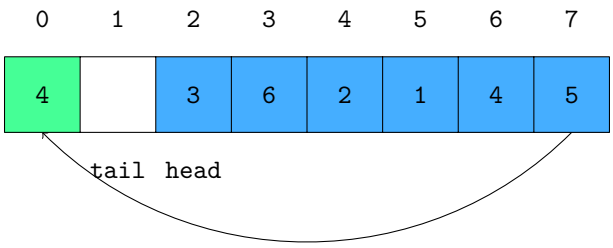


Figura 4.17: Coda piena

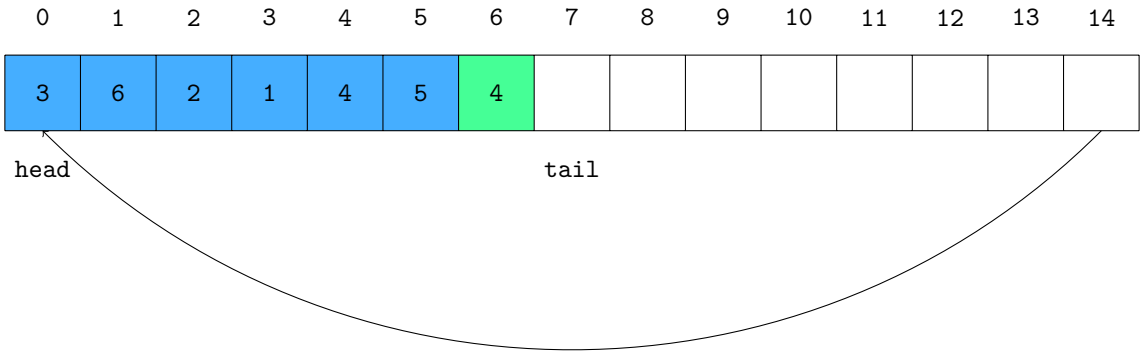


Figura 4.18: Espansione dell'array circolare e linearizzazione degli elementi