

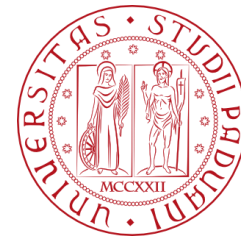
Tutorato 3

15/11/2023

Programmazione ad Oggetti – 2023-2024

Gabriel Rovesti

2103389 – LM Computer Science



UNIVERSITÀ
DEGLI STUDI
DI PADOVA





Conversioni e casting



- Conversioni «safe» (castless)

```
int i=10;  
float f;  
f=i;
```

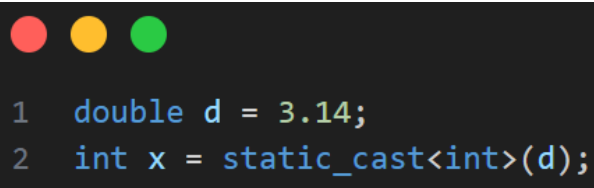
- Narrow conversion* (strette)
- Wide conversion* (larghe)

Narrow conversion		Wide conversion
lose precision 	short	
	int	
	float	
	double	



- `static_cast`

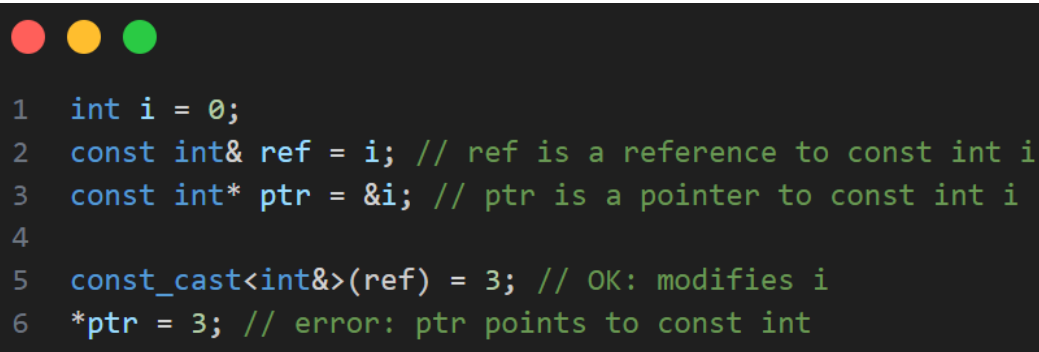
Converte *staticamente* i tipi (non controlla a tempo di esecuzione - runtime) tra loro compatibili (preservando l'indirizzo di memoria). Questo può avvenire in senso *narrow/wide*.



```
1 double d = 3.14;  
2 int x = static_cast<int>(d);
```

- `const_cast`

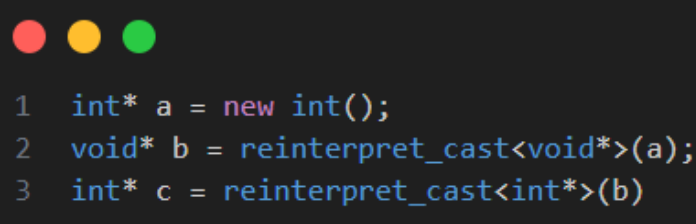
Rimuove il *const* da riferimenti o puntatori che si riferiscono ad oggetti non costanti



```
1 int i = 0;  
2 const int& ref = i; // ref is a reference to const int i  
3 const int* ptr = &i; // ptr is a pointer to const int i  
4  
5 const_cast<int&>(ref) = 3; // OK: modifies i  
6 *ptr = 3; // error: ptr points to const int
```

- `reinterpret_cast`

Forza conversioni tra tipi non correlati, dimenticando il tipo originale e forzando verso il nuovo tipo (*reinterpretando a prescindere il suo valore*)



```
1  int* a = new int();  
2  void* b = reinterpret_cast<void*>(a);  
3  int* c = reinterpret_cast<int*>(b)
```

- `dynamic_cast` (lo vedremo!)

Usato per i tipi *polimorfi* (cambiano forma a seconda del contesto = ereditarietà), usati di solito per convertire «verso il basso» in gerarchia o sullo stesso livello

Template: definizioni



Descrizione di un metodo (o modello) per generare istanze di una funzione che non dipendono dal tipo di argomento.

Essenzialmente, modella il tipo sottostante a prescindere dal suo valore, sostituendo il tipo (*typename*) con la classe desiderata

```
1  template <class T> // <typename T>
2  // T è un tipo generico, detto anche segnaposto
3  T somma(T a, T b) {
4      return a + b;
5  }
6
7  int main() {
8      int a = 1, b = 2;
9      cout << somma<int>(a, b) << endl;
10     cout << somma(a, b) << endl;
11
12     double c = 1.5, d = 2.5;
13     cout << somma<double>(c, d) << endl;
14     cout << somma(c, d) << endl;
15
16     return 0;
17 }
```



Template: definizioni



I template *deducono* il tipo di argomenti esaminando tutti i parametri attuali passati al template da sinistra verso destra (e deve essere esattamente lo stesso).

```
1  int main(){
2      int i; double d, e;
3
4      e = min(i, d);
5      // Non compila perché si
6      // deducono due tipi diversi: int e double
7  }
```

La costruzione per parametri è detta *istanziazione del template*.
Essa può essere implicita (esempio sopra, non specifico i tipi e vengono dedotti)
oppure esplicita

```
1  int main(){
2      int i; double d, e;
3
4      e = min<double>(i, d);
5      // Compila perché converte i in double
6  }
```



Template: definizioni



Ogni metodo o classe parte di template di classe deve avere la dichiarazione apposita sovrastante.

```
1 //fa parte di template e lo specifico
2 template<class T, int size>
3 // dichiarazione esplicita di parametro valore
4 T min (T a, T b) {
5     return a < b ? a : b;
6 }
```

Generalmente i template sono *compilati per separazione*, dichiarando istanze multiple a seconda del contesto.

Hanno alcuni svantaggi:

- rendono difficile il debug
- non hanno information hiding
- non sono supportati da tutti i compilatori



Template di classe



Un uso classico è il *template di classe*.

Piuttosto che definire istanze multiple di una classe (sx), ne definiamo una (dx).

```
1  class QueueInt{
2      public:
3          Queue();
4          ~Queue();
5          bool isEmpty() const;
6          void add(const int& newItem);
7          int remove();
8  };
9
10 class QueueString{
11     public:
12         Queue();
13         ~Queue();
14         bool isEmpty() const;
15         void add(const string& newItem);
16         string remove();
17 };
18
```

```
1  template<class T>
2  class Queue{
3      public:
4          Queue();
5          ~Queue();
6          void push(T);
7          T pop();
8          bool isEmpty();
9  };

```



Template di classe



```
1  template<class T>
2  class Queue{
3  public:
4      Queue();
5      ~Queue();
6      void push(T);
7      T pop();
8      bool isEmpty();
9  private:
10     QueueItem<T> *head;
11     QueueItem<T> *tail;
12 };
```

```
1  template<class T>
2  class QueueItem{
3  public:
4      QueueItem(T);
5      ~QueueItem();
6      T value;
7      QueueItem<T> *next;
8  };
```



Metodi di template

I metodi di template possono essere *inline* (sx) oppure *definiti esternamente* (dx).

```
1  template<class T>
2  class Queue{
3  public:
4      Queue();
5      ~Queue();
6      void push(T);
7      T pop();
8      bool isEmpty();
9  private:
10     QueueItem<T> *head;
11     QueueItem<T> *tail;
12     //inline
13     Queue(): head(nullptr), tail(nullptr){};
14 };
```

```
1  template<class T>
2  Queue<T>::Queue(){
3      head = tail = 0;
4  }
5
```

Vengono usati se e solo se il programma usa effettivamente quei metodi

- Friend di una classe/funzione non template
 - Non fa parte dello stesso template, lo dichiaro amico
 - Diventa amico di tutte le istanze del template di classe C
- Friend di un template di classe C /funzione associato
 - Contiene alcuni parametri del template C
 - Diventa amico della sola istanza (classe/funzione) associata al template C
- Friend di un template di classe C /funzione non associato
 - I template contengono tra di loro parametri diversi
 - Non supportano dichiarazioni friend

- Caso particolare: operatore di stampa

Viene definito come template di funzione esterna per accedere ai campi privati.

```
1  template<class T>
2  ostream& operator<<(ostream& os, const Queue<T>& v) {
3      os << "[";
4      QueueItem<T>* p = v.head; // per amicizia con Queue
5      for (int i = 0; i < v.size(); ++i) { //per amicizia con QueueItem
6          os << p->info << " ";
7          p = p->next;
8      }
9      return os;
10 }
```

Per fare ciò, occorre dichiararlo *friend* nelle classi interessate:

```
1  friend ostream& operator << <T> (ostream&, const Queue<T>&);
```

Esercizio 1: Cosa Stampa



```
1  class A{
2      public:
3          A(int x=0) {cout << x << "A() ";}
4  }
5
6  template<class T>
7  class C{
8      public:
9          static A s;
10 }
11
12 template <class T>
13 A C<T>::s=A();
14
15 int main(){
16     C<double> c;
17     C<int> d;
18     C<int>::s = A(2);
19 }
```



Esercizio 1: Soluzione



```
1  class A{
2      public:
3          A(int x=0) {cout << x << "A() ";}
4      }
5
6      template<class T>
7      class C{
8          public:
9              static A s;
10     }
11
12     template <class T>
13     A C<T>::s=A();
14
15     int main(){
16         C<double> c;
17         C<int> d;
18         C<int>::s = A(2);
19     }
```

```
1  // 0A() 2A()
```



Esercizio 2: Cosa Stampa



```
1  #include <iostream>
2  #include <string>
3
4  using namespace std;
5
6  template <class T>
7  class C{
8      template <class V>
9      friend void fun(C<V>);
10     private:
11         T x;
12     public:
13         C(T y): x(y) {}
14 };
15
```

```
1  template<class T>
2  void fun(C<T> t){
3      cout << t.x << " ";
4      C<double> c(3.1);
5      cout << c.x << endl;
6  }
7
8  int main(){
9      C<int> c(4);
10     C<string> s("blob");
11     fun(c);
12     fun(s);
13 }
```



Esercizio 2: Soluzione

```
1  #include <iostream>
2  #include <string>
3
4  using namespace std;
5
6  template <class T>
7  class C{
8      template <class V>
9      friend void fun(C<V>);
10     private:
11         T x;
12     public:
13         C(T y): x(y) {}
14 };
15
```

```
1  template<class T>
2  void fun(C<T> t){
3      cout << t.x << " ";
4      C<double> c(3.1);
5      cout << c.x << endl;
6  }
7
8  int main(){
9      C<int> c(4);
10     C<string> s("blob");
11     fun(c);
12     fun(s);
13 }
```

```
1  /*
2  Stampa: 4 3.1 - istanziazione implicita fun<int>
3  Stampa: blob 3.1 - istanziazione implicita fun<string>
4  */
```


Template di classe annidati



UNIVERSITÀ
DEGLI STUDI
DI PADOVA

```
1  template<class T>
2  class Queue{
3      private:
4          class QueueItem{
5              public:
6                  T value;
7                  QueueItem* next;
8                  QueueItem(T value, QueueItem* next): value(value), next(next){}
9              };
10     }
11 };
```



Esercizio 3: Compila/Non compila



```
1 //dichiarazione incompleta
2 template<class T> class D;
3
4 template<class T1, class T2>
5 class C{
6     //amicizia associata
7     friend class D<T1>;
8     private;
9     T1 t1; T2 t2;
10 };
11
```

```
1 int main(){
2     D<char> d; d.m();
3     D<char> d; d.n();
4     D<char> d; d.o();
5     D<char> d; d.p();
6     D<char> d; d.q();
7     D<char> d; d.r();
8 }
```

```
1 template <class T>
2 class D{
3     public:
4         void m(){C<T, T> c;
5             cout << c.t1 << c.t2;
6         }
7         void n(){C<Tint, T> c;
8             cout << c.t1 << c.t2;
9         }
10        void o(){C<T, int> c;
11            cout << c.t1 << c.t2;
12        }
13        void p(){C<int, int> c;
14            cout << c.t1 << c.t2;
15        }
16        void q(){C<int, double> c;
17            cout << c.t1 << c.t2;
18        }
19        void r(){C<char, double> c;
20            cout << c.t1 << c.t2;
21        }
22 };
```



Esercizio 3: Soluzione

```
1 //dichiarazione incompleta
2 template<class T> class D;
3
4 template<class T1, class T2>
5 class C{
6     //amicizia associata
7     friend class D<T1>;
8     private;
9     T1 t1; T2 t2;
10 };
11
```

```
1 int main(){
2     D<char> d; d.m();
3     D<char> d; d.n();
4     D<char> d; d.o();
5     D<char> d; d.p();
6     D<char> d; d.q();
7     D<char> d; d.r();
8 }
```

```
1 /*
2 Compila
3 Non compila
4 Compila
5 Non compila
6 Non compila
7 Compila
8 */
```

```
1 template <class T>
2 class D{
3     public:
4         void m(){C<T, T> c;
5             cout << c.t1 << c.t2;
6         }
7         void n(){C<Tint, T> c;
8             cout << c.t1 << c.t2;
9         }
10        void o(){C<T, int> c;
11            cout << c.t1 << c.t2;
12        }
13        void p(){C<int, int> c;
14            cout << c.t1 << c.t2;
15        }
16        void q(){C<int, double> c;
17            cout << c.t1 << c.t2;
18        }
19        void r(){C<char, double> c;
20            cout << c.t1 << c.t2;
21        }
22 };
```

Esercizio 4: Template

```
1  /*
2  Si considerino le seguenti definizioni. Fornire una dichiarazione
3  (non e` richiesta la definizione) dei membri pubblici della classe Z
4  nel minor numero possibile in modo tale che la compilazione del
5  main() non produca errori. Attenzione: ogni dichiarazione in Z
6  non necessaria per la corretta compilazione del main() e'
7  penalizzata.
8  */
9
10 class Z {
11     ...
12 };
13 template <class T1, class T2 =Z>
14 class C {
15 public:
16     T1 x;
17     T2* p;
18 };
```

```
1  template<class T1,class T2>
2  void fun(C<T1,T2>* q) {
3      ++(q->p);
4      if(true == false) cout << ++(q->x);
5      else cout << q->p;
6      (q->x)++;
7      if(*(q->p) == q->x) *(q->p) = q->x;
8      T1* ptr = &(q->x);
9      T2 t2 = q->x;
10 }
11 int main(){
12     C<Z> c1; fun(&c1); C<int> c2; fun(&c2);
13 }
```

Esercizio 4: Soluzione



```
1  template<class T1,class T2>
2  void fun(C<T1,T2>* q) {
3      ++(q->p); //nessun requirement
4      if(true == false) cout << ++(q->x); else cout << q->p; //q->x di tipo T1, operator++() T1
5      (q->x)++; //operator++(int) su T1
6      if(*(q->p) == q->x) *(q->p) = q->x; //(1) bool operator==(T2, T1), (2) operator=(T2, const T1&)
7      T1* ptr = &(q->x); //nessun requirement
8      T2 t2 = q->x; //T2(const T1&)
9  }
10
11 class Z {
12     public:
13     int& operator++();
14     int operator++(int);
15     bool operator==(const Z&) const;
16     Z (const int&); //converte int a Z
17 };
```



In the C++ programming language, the **C++ Standard Library** is a collection of **classes** and **functions**, which are written in the **core language** and part of the C++ **ISO Standard** itself.^[1]

- Rappresenta un insieme di classi template (STL = Standard Template Library) in grado di fornire strutture dati, funzioni ed algoritmi. Precisamente contiene:
 1. *Containers* (contenitori per accesso a classi e dati)
 2. *Algorithms* (algoritmi su insiemi/range di elementi)
 3. *Iterators* (iteratori per lavorare su sequenze di valori)
 4. *Functors* (funtori, lavorano sugli oggetti invocati dalle funzioni per overloading)

Contenitori libreria STL



UNIVERSITÀ
DEGLI STUDI
DI PADOVA

C++ library headers				
<algorithm>	<iomanip>	<list>	<queue>	<string>
<bitset>	<ios>	<locale>	<set>	<stringstream>
<complex>	<iosfwd>	<map>	<sstream>	<typeinfo>
<deque>	<iostream>	<memory>	<stack>	<utility>
<exception>	<istream>	<new>	<stdexcept>	<valarray>
<fstream>	<iterator>	<numeric>	<streambuf>	<vector>
<functional>	<limits>	<ostream>		
Headers added in C++11				
<array>	<condition_variable>	<mutex>	<scoped_allocator>	<type_traits>
<atomic>	<forward_list>	<random>	<system_error>	<typeindex>
<chrono>	<future>	<ratio>	<thread>	<unordered_map>
<codecvt>	<initializer_list>	<regex>	<tuple>	<unordered_set>
Headers added in C++14				
<shared_mutex>				
Headers added in C++17				
<any>	<filesystem>	<optional>	<string_view>	<variant>
<execution>	<memory_resource>			
Headers added in C++20				
<barrier>	<concepts>	<latch>	<semaphore>	<stop_token>
<bit>	<coroutine>	<numbers>	<source_location>	<syncstream>
<charconv>	<format>	<ranges>		<version>
<compare>				
Headers added in C++23				
<expected>	<flat_set>	<mdspan>	<spanstream>	<stdfloat>
<flat_map>	<generator>	<print>	<stacktrace>	



- Ogni classe container ha tra i suoi membri i seguenti tipi:
 - `C::value_type` → Tipo degli oggetti memorizzati nel container
 - `C::iterator` → Iteratore sui singoli elementi
 - `C::const_iterator` → Iteratore costante per accedere agli elementi senza modificarli

Ogni classe container ha tra i suoi membri i seguenti costruttori:

- `C(const C&)` : ridefinizione del costruttore di copia
- `~C()` : ridefinizione del distruttore
- `C& operator=` : ridefinizione dell'operatore di assegnazione

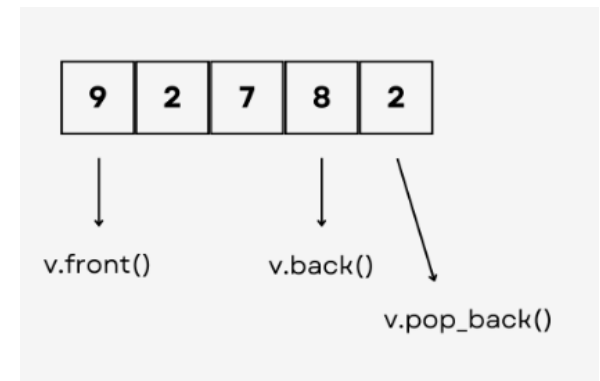
Ogni classe container ha tra i suoi membri i seguenti metodi:

- `size_type size()` : dimensione del contenitore
- `empty()` : controllo se vuoto
- `size_type max_size()` : massima dimensione del contenitore

- Alcuni esempi di classi comuni:
- Contenitori di sequenza (hanno un ordine, cioè hanno front e back)
 - **vector**
 - **list**
 - **deque**
- Contenitori associativi (non più con un ordine, ma hanno associazioni uniche tra gli elementi)
 - **set**
 - **multiset**
 - **map**
 - **multimap**
- Il più semplice e il più efficiente è **vector**
- Contenitori adapters (adattano il tipo di dato sottostante ad un diverso tipo di funzionalità)
 - **stack**
 - **queue**

Caratteristiche di **vector**:

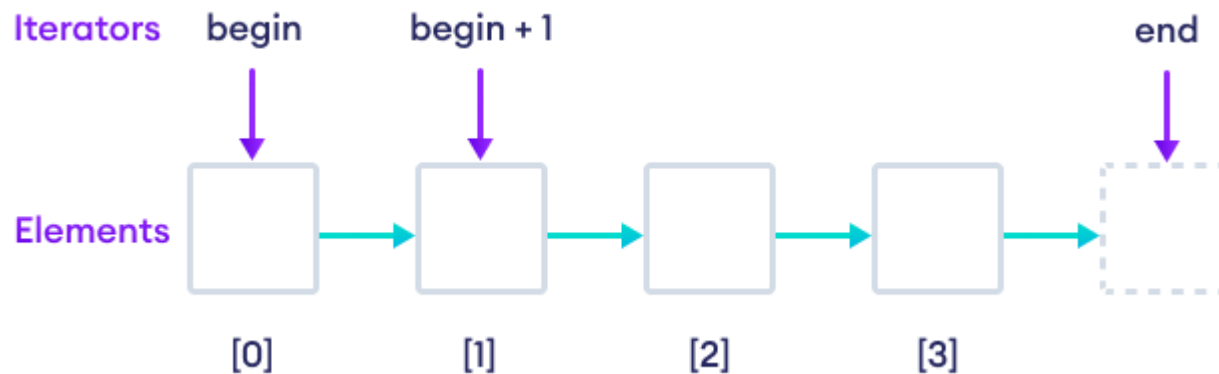
1. Contenitore di sequenza che supporta l'accesso casuale agli elementi (tempo $O(1)$)
2. Inserimento e rimozione in coda in tempo ammortizzato costante
3. Inserimento e rimozione arbitraria in tempo *lineare ammortizzato*
4. Capacità di un vector dinamicamente variabile
5. Gestione della memoria automatica



```
1 //Dichiarazione
2 vector<int> v(10);
3
4 //Accesso ai singoli elementi
5 int n = 5;
6 vector<int> v1(n);
7 int x = v1[0];
8 for(int i=0; i<n; i++) v1[i] = i;
```

```
1 //size() - ritorna numero elementi vector
2 //capacity() - ritorna numero elementi allocati
3
4 // push_back() - aggiunge elemento in coda con costruttore di copia
5 vector<int> v;
6 while(cin >> i) v.push_back(i);
```

- Utilizzati per attraversare i container...



- but keep in mind the const-correctness!*

iterator vs const_iterator

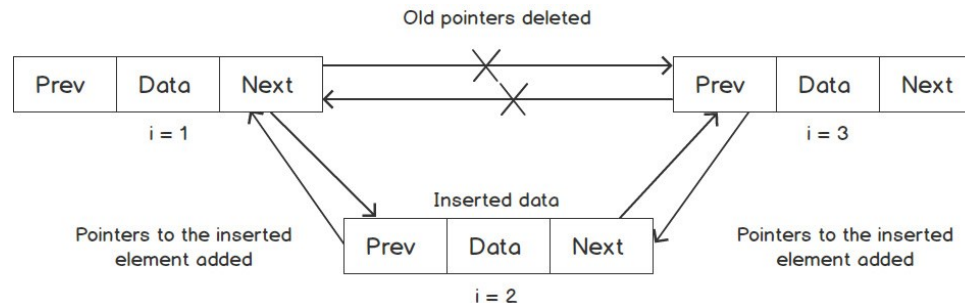
- Caratteristiche di **iterator**:
 - Sono utilizzati per attraversare una sequenza e possono essere utilizzati per leggere o scrivere i valori degli elementi.
 - Non sono costanti, il che significa che è possibile modificarli o utilizzarli per modificare gli elementi nella sequenza
- Caratteristiche di **const_iterator**:
 - Sono utilizzati per attraversare una sequenza, ma sono costanti e non è possibile utilizzarli per modificare gli elementi nella sequenza.
 - Tuttavia, possono essere utilizzati per leggere il valore degli elementi in modo sicuro. Questo è utile quando si desidera assicurarsi che la sequenza rimanga invariata durante l'iterazione.

- Sono bidirezionali e agiscono su determinate posizioni del vettore. Quali
 - **begin()** – accesso al primo elemento
 - **end()** – accesso all'ultimo elemento
- Vi sono alcuni altri metodi di inserimento e di rimozione:
 - **insert()** – inserimento in una posizione precisa
 - Può risultare inefficiente e dipende dall'implementazione
 - Generalmente, tutti gli elementi *past the end* (dopo la fine)
 - **push_front()** – inserimento davanti
 - **push_back()** – inserimento dietro
 - **pop_front()** – rimozione primo elemento
 - **pop_back()** – rimozione ultimo elemento
 - **erase()** – rimozione in una posizione precisa
 - **clear()** – cancellazione dall'inizio alla fine

- È implementata come *doubly-linked list* (lista doppiamente collegata, ogni nodo punta al precedente e al successivo)
- Contenitore sequenza che supporta inserimento/rimozione di elementi all'inizio/alla metà/alla fine della lista in tempo costante

Conseguenze:

- Inserimento e rimozione più efficienti di **vector**
- Accesso agli elementi meno efficiente di **vector** (devo sempre percorrere la lista dall'inizio alla fine)



list vs vector



UNIVERSITÀ
DEGLI STUDI
DI PADOVA

Default data structure to think of in C++ is the **Vector**.

Consider the following points...

1] Traversal:

List nodes are scattered everywhere in memory and hence list traversal leads to **cache misses**. But traversal of vectors is smooth.

2] Insertion and Deletion:

Average 50% of elements must be shifted when you do that to a Vector but caches are very good at that! But, with lists, you need to **traverse to the point of insertion/deletion...** so again... cache misses! And surprisingly vectors win this case as well!

3] Storage:

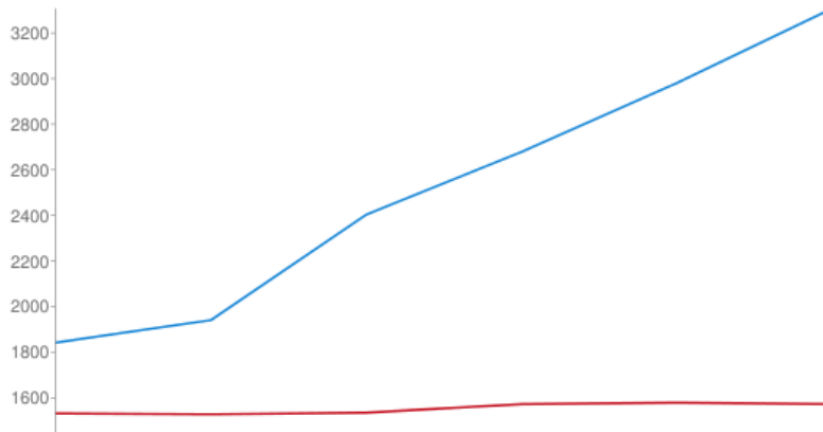
When you use lists, there are 2 pointers per elements(forward & backward) so a List is much bigger than a Vector! Vectors need just a little more memory than the actual elements need.



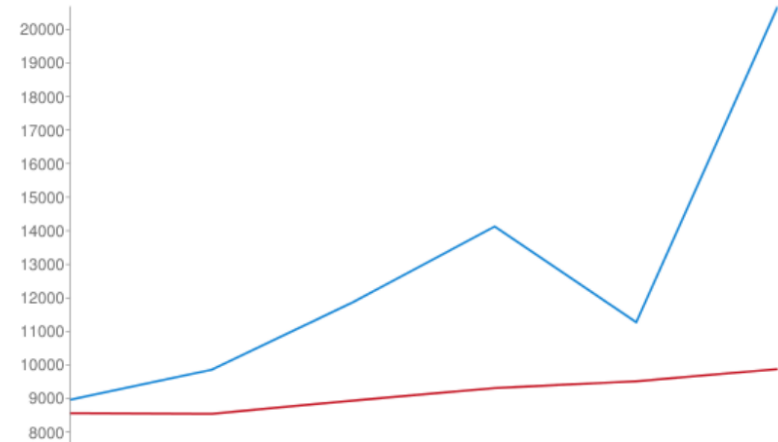
Where vector wins



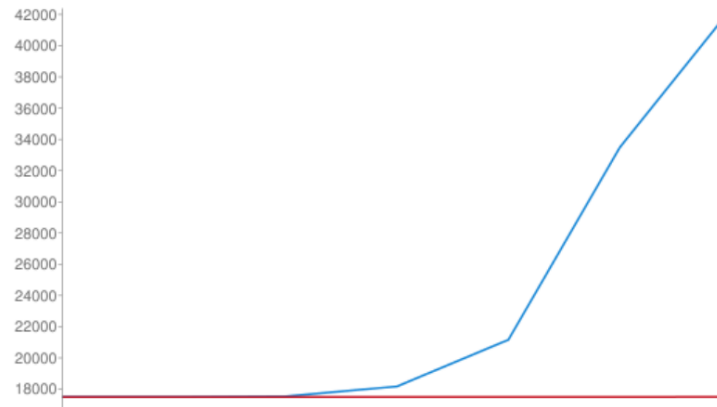
insert (1024 bytes) - milliseconds (less is better)



remove (1024 bytes) - milliseconds (less is better)



push front (8 bytes) - milliseconds (less is better)

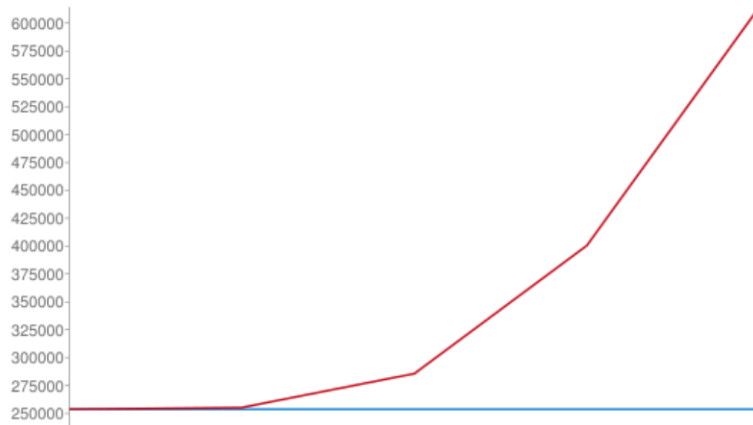


Where list wins

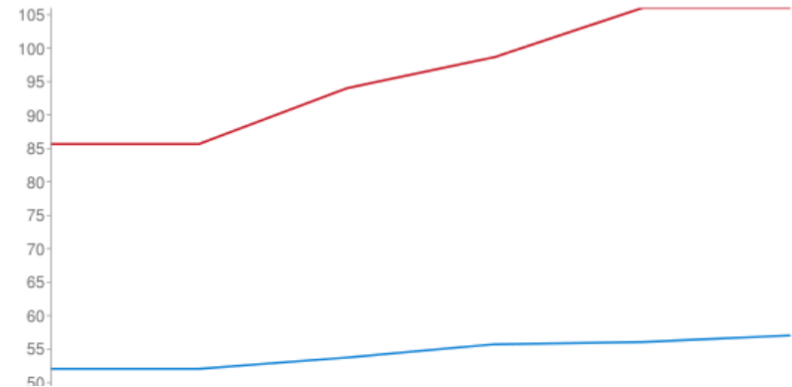


Note: random insert

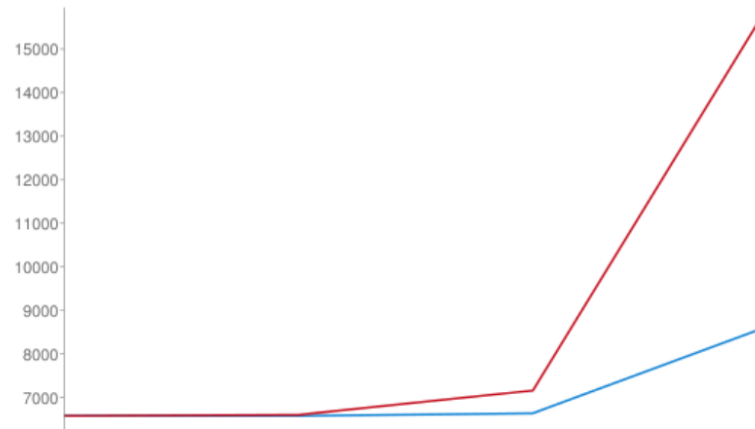
find (8 bytes) - microseconds (less is better)



insert (8 bytes) - milliseconds (less is better)



sort (8 bytes) - milliseconds (less is better)



```
#include <algorithm>
```

Non-Modifying Queries

- finding elements / existence queries
- minimum / maximum
- comparing ranges of elements
- binary search of sorted ranges

Modifying Operations

- copying / moving elements
- replacing / transforming elements
- removing elements
- union/intersection/etc. of sorted ranges

```
1  #include <iostream>
2  #include <algorithm> //necessario
3  #include <vector>
4  using namespace std;
5
6  int main()
7  {
8      vector<int> v;
9      v.push_back(1);
10     v.push_back(3);
11     v.push_back(2);
12     v.push_back(4);
13     v.push_back(5);
14
15     for_each(v.begin(), v.end(), [](int n) { cout << n << " "; });
16     cout << endl;
```

Esercizio 5: Modellazione



Definire un template di classe contenitore `Dizionario<T>` i cui oggetti rappresentano una collezione di coppie (chiave, valore) dove la chiave è una stringa ed il valore è di tipo `T`. Ad una certa stringa può essere associato un solo valore di `T`. Si tratta quindi di definire un template di classe analogo al contenitore STL `map<string, T>`. Dovranno essere disponibili le seguenti funzionalità:

- inserimento: `bool insert(string, const T&)`
- rimozione: `bool erase(string)`
- ricerca per chiave: `T* findValue(string)`
- ricerca per valore: `vector<string> findKey(const T&)`
- overloading degli operatori di indicizzazione e output

