



Laurea triennale in Informatica

modulo (CFU 6) di

Programmazione II e Lab.

prof. Mariarosaria Rizzardi

Centro Direzionale di Napoli – Isola C4

stanza: n. 423 – IV piano Lato Nord

tel.: 081 547 6545

email: mariarosaria.rizzardi@uniparthenope.it

Programmazione in C++:

- le **classi template** per ADT pila, coda, lista lineare, lista bidirezionale

La Libreria Standard del C++ (**STL - Standard Template Library**), progettata dallo stesso inventore del C++ Bjarne Stroustrup, mette a disposizione numerose **classi template**, alcune delle quali per le strutture dati di base, e molte **funzioni generiche** che implementano algoritmi fondamentali.

In particolare per le strutture dati dinamiche lineari ci sono (fra altre) le **classi template**:

- **stack** per la ADT pila;
- **queue** per la ADT coda;
- **forward_list** per la ADT lista lineare;
- **list** per la ADT lista bidirezionale.

ADT: Abstract Data Type

std::stack<T>: Esempio 1a

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

void showstack(stack<int> s);

int main() {
    stack<int> s;
    cout << "\ns.size() = " << s.size();
    if ( s.empty() )
        cout << "\tstack vuoto\n";
    else
        cout << "\tstack non vuoto\n";
    s.push(10);
    cout << "s.push(10) ==> Lo stack s contiene:"; showstack(s);
    s.push(30);
    cout << "s.push(30) ==> Lo stack s contiene:"; showstack(s);
    s.push(20);
    cout << "s.push(20) ==> Lo stack s contiene:"; showstack(s);
    s.push(5);
    cout << "s.push(5) ==> Lo stack s contiene:"; showstack(s);
    s.push(1);
    cout << "s.push(1) ==> Lo stack s contiene:"; showstack(s);


    cout << "\ns.size() = " << s.size();
    if ( !s.empty() )
    {
        cout << "\ns.top() = " << s.top();
        s.pop();
        cout << "\ns.pop() ==> Lo stack s contiene:"; showstack(s);
    }
    return 0;
}
```

...ardi

std::stack<T>: Esempio 1a (cont.)

```
void showstack(stack<int> s)
{
    while ( !s.empty() )
    {
        cout << '\t' << s.top();
        s.pop();
    }
    cout << '\n';
}
```

Perché, nonostante venga usato il metodo **pop()**, lo stack incrementa comunque il suo contenuto?



```
s.size() = 0      stack vuoto
s.push(10) ==> Lo stack s contiene:  10
s.push(30) ==> Lo stack s contiene:  30  10
s.push(20) ==> Lo stack s contiene:  20  30  10
s.push(5)  ==> Lo stack s contiene:  5   20  30  10
s.push(1)  ==> Lo stack s contiene:  1   5   20  30  10
s.size() = 5
s.top()    = 1
s.pop()    ==> Lo stack s contiene:  5   20  30  10
```

std::stack<T>: Esempio 1b

Stesso main()

```
void showstack(stack<int> &s)
{
    while ( !s.empty() )
    {
        cout << '\t' << s.top();
        s.pop();
    }
    cout << '\n';
}
```

Spiegare perché i risultati sono diversi da quelli dell'esempio precedente?

```
s.size() = 0      stack vuoto
s.push(10) ==> Lo stack s contiene:      10
s.push(30) ==> Lo stack s contiene:      30
s.push(20) ==> Lo stack s contiene:      20
s.push(5)  ==> Lo stack s contiene:       5
s.push(1)  ==> Lo stack s contiene:       1

s.size() = 0
```

<http://www.cplusplus.com/reference/stack/stack/>

std::queue<T>: Esempio 2a

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

void showqueue(queue <int> q);

int main() {
    queue <int> q;
    cout << "\nq.size() = " << q.size();
    if ( q.empty() )
        cout << "\tcoda vuota\n";
    else
        cout << "\tcoda non vuota\n";
    q.push(10);
    cout << "q.push(10) ==> La coda q contiene:"; showqueue(q);
    q.push(30);
    cout << "q.push(30) ==> La coda q contiene:"; showqueue(q);
    q.push(20);
    cout << "q.push(20) ==> La coda q contiene:"; showqueue(q);
    q.push(5);
    cout << "q.push(5) ==> La coda q contiene:"; showqueue(q);
    q.push(1);
    cout << "q.push(1) ==> La coda q contiene:"; showqueue(q);


    cout << "\nq.size() = " << q.size();
    if ( !q.empty() )
    {
        cout << "\nq.front() = " << q.front();
        q.pop();
        cout << "\nq.pop() ==> La coda q contiene:"; showqueue(q);
    }
    return 0;
}
```

...ardi

std::queue<T>: Esempio 2a (cont.)

```
void showqueue(queue<int> q)
{
    while ( !q.empty() )
    {
        cout << '\t' << q.front();
        q.pop();
    }
    cout << '\n';
}
```

Perché, nonostante venga usato il metodo **pop()**, la coda incrementa comunque il suo contenuto?



```
q.size() = 0      coda vuota
q.push(10) ==> La coda q contiene: 10
q.push(30) ==> La coda q contiene: 10 30
q.push(20) ==> La coda q contiene: 10 30 20
q.push(5) ==> La coda q contiene: 10 30 20 5
q.push(1) ==> La coda q contiene: 10 30 20 5 1

q.size() = 5
q.front() = 10
q.pop() ==> La coda q contiene: 30 20 5 1
```


std::queue<T>: Esempio 2b

Stesso main()

```
void showqueue(queue<int> &q)
{
    while ( !q.empty() )
    {
        cout << '\t' << q.front();
        q.pop();
    }
    cout << '\n';
}
```

Spiegare perché i risultati sono diversi da quelli dell'esempio precedente?

```
q.size() = 0      coda vuota
q.push(10) ==> La coda q contiene:      10
q.push(30) ==> La coda q contiene:      30
q.push(20) ==> La coda q contiene:      20
q.push(5)  ==> La coda q contiene:       5
q.push(1)  ==> La coda q contiene:       1

q.size() = 0
```

<http://www.cplusplus.com/reference/queue/queue/>

std::forward_list<T>: Esempio 3

```
#include <iostream>
#include <iterator>
#include <forward_list>
using namespace std;
void showForwList(forward_list<int> flist);
```

```
void showForwList(forward_list<int> flist)
{
    for ( int &r : flist )
        cout << '\t' << r;
    cout << '\n';
}
```

```
int main() {
    forward_list<int> flist;
    forward_list<int>::iterator it;
    if ( flist.empty() )
        cout << "\tlista lineare vuota\n";
    else
        cout << "\tlista lineare non vuota\n";
    flist.push_front(10);
    cout << "flist.push_front(10) ==> La lista contiene:"; showForwList(flist);
    flist.push_front(30);
    cout << "flist.push_front(30) ==> La lista contiene:"; showForwList(flist);
    flist.push_front(20);
    cout << "flist.push_front(20) ==> La lista contiene:"; showForwList(flist);
    it=flist.insert_after(flist.begin(),{1,3,5});
    cout << "it=flist.insert_after(flist.begin(),{1,3,5}) ==> La lista contiene:";
    showForwList(flist);

    cout << "*it = " << *it << endl;
    it=flist.insert_after(it,7);
    cout << "it=flist.insert_after(it,7) ==> La lista contiene:"; showForwList(flist);
    cout << "*it = " << *it << endl;
    it=flist.begin(); advance(it,3);
    cout << "it=flist.begin(); advance(it,3); ==> *it = " << *it << endl;
    it=flist.insert_after(it,9);
    cout << "it=flist.insert_after(it,9) ==> La lista contiene:"; showForwList(flist);
    cout << "*it = " << *it << endl;
```

iteratore sulla lista

inserisce in testa

inserisce dopo 1° nodo

inserisce dopo nodo puntato da it

it punta al 3° nodo dopo il 1°

...

std::forward_list<T>: Esempio 3 (cont.)

```
...  
flist.pop_front();  
cout << "flist.pop_front() ==> La lista contiene:"; showForwList(flist);  
it = flist.begin();  
cout << "*it = " << *it << endl;  
flist.erase_after(it,flist.end());  
cout << "flist.erase_after(it,flist.end()) ==> La lista contiene:"; showForwList(flist);  
  
return 0;  
}
```

← elimina nodo di testa

← elimina nodi dopo quello puntato da it e fino alla fine

```
lista lineare vuota  
flist.push_front(10) ==> La lista contiene: 10  
flist.push_front(30) ==> La lista contiene: 30 10  
flist.push_front(20) ==> La lista contiene: 20 30 10  
it=flist.insert_after(flist.begin(),{1,3,5}) ==> La lista contiene: 20 1 3 5 30 10  
*it = 5  
it=flist.insert_after(it,7) ==> La lista contiene: 20 1 3 5 7 30 10  
*it = 7  
it=flist.begin(); advance(it,3); ==> *it = 5  
it=flist.insert_after(it,9) ==> La lista contiene: 20 1 3 5 9 7 30 10  
*it = 9  
flist.pop_front() ==> La lista contiene: 1 3 5 9 7 30 10  
*it = 1  
flist.erase_after(it,flist.end()) ==> La lista contiene: 1
```

std::list<T>: Esempio 4

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

void showForwList(list<int> Blist);
void showBackwList(list<int> Blist);
```

```
void showForwList(list<int> Blist)
{
    /// visita lista in avanti
    list<int> :: iterator it;
    for ( it=Blist.begin(); it!=Blist.end(); it++ )
        cout << '\t' << *it;
    cout << '\n';
}
```

```
int main() {
    list<int> Blist;
    list<int>::iterator it;
    list<int>::reverse_iterator rit;
    if ( Blist.empty() )
        cout << "\tlista bidirezionale vuota\n";
    else
        cout << "\tlista bidirezionale non vuota\n";
    Blist.push_front(10);
    cout << "Blist.push_front(10) ==> La lista bidirezionale contiene forward:";
    showForwList(Blist);
    Blist.push_front(30);
    cout << "Blist.push_front(30) ==> La lista bidirezionale contiene forward:";
    showForwList(Blist);
    Blist.push_back(20);
    cout << "Blist.push_back(20) ==> La lista bidirezionale contiene forward:";
    showForwList(Blist);
    cout << "
    ==> La lista bidirezionale contiene backward:";
    showBackwList(Blist);
    it=Blist.begin(); ++it;
    cout << "it=Blist.begin(); ++it; ==> *it = " << *it << endl;
    it=Blist.insert(it,{1,3,5});
    cout << "it=Blist.insert(it,{1,3,5}) ==> La lista contiene:"; showForwList(Blist);
    cout << "*it = " << *it << endl;
```

← iteratori sulla lista bidirezionale

← inserisce in testa

← inserisce in coda

← it punta al 2° nodo

← inserisce prima del nodo puntato da it

...

std::list<T>: Esempio 4 (cont.)

```
void showbackwList(list<int> Blist)
{    /// visita lista all'indietro
    list<int> :: reverse_iterator rit;
    for ( rit=Blist.rbegin(); rit!=Blist.rend(); rit++ )
        cout << '\t' << *rit;
    cout << '\n';
}
```

```
...
it=Blist.end(); it--;
cout << "it=Blist.end(); it--; ==> *it = " << *it << endl;
rit=Blist.rbegin();
cout << "rit=Blist.rbegin(); ==> *rit = " << *rit << endl;
it=Blist.insert(it,2,15);
cout << "it=Blist.insert(it,2,15) ==> La lista contiene:"; showForwList(Blist);
Blist.pop_front();
cout << "Blist.pop_front() ==> La lista contiene:"; showForwList(Blist);
Blist.pop_back();
cout << "Blist.pop_back() ==> La lista contiene:"; showForwList(Blist);
...
```

Diagrammatic annotations for the code above:

- it punta all'ultimo nodo
- rit punta all'ultimo nodo
- inserisce {15,15} prima del nodo puntato da it
- elimina nodo di testa
- elimina nodo di coda

```
lista bidirezionale vuota
Blist.push_front(10) ==> La lista bidirezionale contiene forward: 10
Blist.push_front(30) ==> La lista bidirezionale contiene forward: 30 10
Blist.push_back(20) ==> La lista bidirezionale contiene forward: 30 10 20
                        ==> La lista bidirezionale contiene backward: 20 10 30

it=Blist.begin(); ++it; ==> *it = 10
it=Blist.insert(it,{1,3,5}) ==> La lista contiene: 30 1 3 5 10 20
*it = 1

it=Blist.end(); it--; ==> *it = 20
rit=Blist.rbegin(); ==> *rit = 20
it=Blist.insert(it,2,15) ==> La lista contiene: 30 1 3 5 10 15 15 20
Blist.pop_front() ==> La lista contiene: 1 3 5 10 15 15 20
Blist.pop_back() ==> La lista contiene: 1 3 5 10 15 15
```

std::list<T>: Esempio 4 (cont.)

it punta oltre l'ultimo nodo

arretra di 3 nodi

```
...  
it=Blist.end();  
advance(it,-3);  
cout << "it=Blist.end(); advance(it,-3); ==> *it = " << *it << endl;  
  
return 0;  
}
```

it=Blist.end();

```
...  
Blist.pop_back() ==> La lista contiene:  
it=Blist.end(); advance(it,-3); ==> *it = 10
```

advance(it,-3);

<http://www.cplusplus.com/reference/list/list/>