CINECA Consorzio Interuniversitario

C++ STANDARD LIBRARY

Marzia Rivi
m.rivi@cineca.it

CINECA Consorzio Interuniversitario

Cosa fornisce

1. Libreria di funzioni indipendenti di utilizzo generale ereditate dal C:
   - Funzioni di I/O basate sul C
   - Gestione di stringhe C e caratteri
   - Funzioni matematiche
   - Funzioni per la localizzazione e la gestione della memoria
   - Funzioni di servizio e di supporto al linguaggio
2. Libreria di classi per la programmazione orientata agli oggetti
   - I/O streams
   - Classe String, classi numeriche
   - Gestione delle eccezioni
   - Standard Template Library (STL)

CINECA Consorzio Interuniversitario

Come usarla

#include ...

i file include contengono:

- template di funzioni
- classi
- funzioni generiche (devono essere compilate con il "tipo" istanziato dall'utente)
- funzioni inline
- costanti

includere solo il minimo indispensabile

Tutte le "facility" della Standard Library *sono all'interno* del namespace "std":

- std::sort();
- using std::sort();
- using namespace std;

in fase di link (gestito automaticamente)

CINECA Consorzio Interuniversitario

LANGUAGE SUPPORT	
<limits>	numeric limits
<climits>	C-style numeric scalar-limits macros
<cfloat>	C-style numeric floating-point limit macros
<new>	dynamic memory management
<typeinfo>	run-time type identification support
<exception>	exception-handling support
<cstddef>	C library language support
<csetjmp>	C-style stack unwinding
<cstdlib>	program termination
<ctime>	system clock
<csignal>	C-style signal handling

DIAGNOSTICS

<exception>	***exception class***
<stdexcept>	***standard exceptions***
<cassert>	assert macro
<cerrno>	C-style error handling

LOCALIZATION

<locale>	represent cultural differences
<clocale>	idem, C-style

CINECA Consorzio Interuniversitario

INPUT/OUTPUT

<iosfwd>	forward declarations of I/O facilities
<iostream>	***standard iostream objects and operations***
<ios>	iostream bases
<streambuf>	stream buffers
<istream>	***input stream template***
<ostream>	***ouput stream template***
<iomanip>	***manipulators***
<sstream>	***streams to/from strings***
<cstdlib>	character classification functions
<fstream>	***streams to/from files***
<cstdio>	printf() family I/O
<cwchar>	printf()-style I/O of wide characters

GENERAL UTILITIES:

<utility>	operators and pairs
<functional>	function objects
<memory>	allocators for containers
<ctime>	C-style date and time

CINECA Consorzio Interuniversitario

NUMERICS

<complex>	complex numbers and operations
<valarray>	numeric vectors and operations
<numeric>	generalized numeric operations
<cmath>	standard mathematical functions
<cstdlib>	C-style random numbers, abs(), fabs(), div()

CINECA Consorzio Interuniversitario

STRINGS

<string>	*string* (container)
<cctype>	character classification
<cwctype>	wide-character classification
<cstring>	C-style string functions: strlen(), strcpy(),...
<cwchar>	C-style wide-character string functions
<cstdlib>	C-style string functions: atoi(), atof()...

CINECA Consorzio Interuniversitario

String

Oltre alle stringhe *C-like,* ovvero agli *array* di *char,* la libreria standard C++ mette a disposizione una string dinamica, che cerca di andare incontro alle più varie esigenze: la classe *string*.

```
string first_name = "Mario";
string second_name = "Rossi";
string title = "dr.";
string signature = title + " " + first_name + " "
                        + second_name;
```

Spesso è necessario effettuare poche operazioni su stringhe... e spesso il compito è impegnativo, e soggetto ad errori.
string mette a disposizione uno strumento flessibile ed efficace.

USARE STRING AL POSTO DELLE STRINGHE C-style!

10

CINECA Consorzio Interuniversitario

Constructors

```
string();
string(const string &s, size_type pos=0,
size_type n=npos);
string(const Ch *p, size_type n);
string(const Ch *);
string(size_type n, Ch c);
template<class In> string(In first, In last);

 ~string();

static const size_type npos;
```

CINECA Consorzio Interuniversitario

ESEMPI:

```
string s0;      // empty string
string s00=""; // empty string
string s1='a'; //ERROR - no conversion from char to string
string s2=7;   //ERROR - no conversion from int
string s3(7);  //ERROR - no constructor with int argument
string s4(7,'a');  // "aaaaaaa"
string s5="Frodo"; // "Frodo" copia la C-string
string s6=s5;       // "Frodo" copia string-to-string
string s7(s5,2,3); // "odo" 3 caratteri da s5[2]
string s8("pistacchio",6,3);  // "chi": char* => string…
string s8(string("pistacchio"),6,3) // …così
```

Usando range fuori stringa viene lanciata un'eccezione "out_of_range".

CINECA Consorzio Interuniversitario

Assegnamenti

A differenza del C, dove l'assegnamento di un char* non fa altro che cambiare il puntatore (è necessario usare funzioni come strcpy() per copiare stringhe di caratteri), in string l'assegnamento è una effettiva copia dei contenuti.

```
string s1="verde";
string s2="rosso";
s2=s1; // due copie di "verde"
s1[2] = 'n';  // s1: "vende"  s2: "verde"
string s='a';  // ERRORE non si puo' inizializzare da char
s='a';         // OK: l'assignment e' invece permesso
s="ciao";      // OK.
```

CINECA Consorzio Interuniversitario

concatenation

```
string operator+(const string &s1, const string &s2); //string+string
string operator+(const Ch *p, const string &s2); // C-string + string
string operator+(const Ch c, const string &s2); // char + string
string operator+(const string &s2, const Ch *p); // string + C-string
string operator+(const string &s2, const Ch c); // string + char
```

Il concatenamento avviene tramite operatore "+":

```
string s1="voi";
string s2 = "ciao " + "a" + " tutti"; // ERRORE!
string s2 = "ciao " + s1 + "tutti";  // OK
string s2 = "ciao " + string("a") + "tutti";  // OK
string s2 = string("ciao ") + "a" + "tutti";  // OK
string s2 = "ciao " + "a" + string("tutti");  // ERRORE!
```

CINECA Consorzio Interuniversitario

append

```
s1.append(s2)   <->   s1 += s2
```

```
string append(const &string);
string append(const &string, size_type pos,
size_type n);
string append(const Ch *p, size_type n);
string append(const Ch *p);
string append(size_type n, Ch c);
template<class In> append(In first, In
last);
```

CINECA Consorzio Interuniversitario

substring

```
string substr(size_type i=0, size_type n=npos);
```

La maggior parte dei metodi di string permettono di operare su substring definite da posizione iniziale (default, 0) e lunghezza (default, npos - tutta):

se non è indicato, di default s.substr() == s

In mancanza del metodo "esplicito" è possibile usare il metodo substr().

CINECA Consorzio Interuniversitario

insert

```
string insert(size_type pos, const &string);
string insert(size_type pos, const &string, size_type
pos2, size_type n);
string insert(size_type pos, const Ch *p, size_type n);
string insert(size_type pos, const Ch *p);
string insert(size_type pos, size_type n, const Ch c);

iterator  insert(iterator p, Ch c);
void insert(iterator p, size_type n Ch c);
template<class In> void insert(iterator p, In first, In
last);
```

CINECA Consorzio Interuniversitario

replace

```
string &replace(size_type i, size_type n, const string &s);
string &replace(size_type i, size_type n, const string &s,
size_type i2, size_type n2);
string &replace(size_type i, size_type n, const Ch* p,
size_type n2);
string &replace(size_type i, size_type n, const Ch* p);
string &replace(size_type i, size_type n, size_type n2, Ch
c);

string &replace(iterator i, iterator i2, const string &s);
string &replace(iterator i, iterator i2, const Ch *p,
size_type n);
string &replace(iterator i, iterator i2, const Ch *p);
string &replace(iterator i, iterator i2, size_type n, Ch c);
template<class In> string &replace(iterator i, iterator i2,
In j, In j2);
```

CINECA Consorzio Interuniversitario

erase

"replace with nothing"

```
string &erase(size_type i=0, size_type
n=npos);
string &erase(iterator i);
string &erase(iterator first, iterator last);
```

CINECA Consorzio Interuniversitario

find

```
size_type find(const string &s, size_type i=0) const;
size_type find(const Ch *p, size_type i, size_type n)
const;
size_type find(const Ch *p, size_type i=0) const;
size_type find(Ch c, size_type i=0) const;
```

`string npos;` // è la costante che indica "non trovato": npos è la posizione dopo l'ultimo carattere: one-past-the-end

Set identici di 4 operazioni per:

```
rfind()
find_first_of()
find_last_of()
find_first_not_of()
find_last_not_of()
```

Confronti

gli operatori ==, <=,>=, <,> != eseguono i classici confronti lexicografici: come strcmp() del C, ma MOLTO più leggibile!

int x=s.compare(s2): x=0 se sono uguali, x<0 se s e' lexicograficamente prima di s2 e x>0 altrimenti.

```
int compare(const string &s) const;
int compare(const Ch *p) const;
int compare(size_type pos, size_type n, const string
   &s) const;
int compare(size_type pos, size_type n, const string
   &s, size_type pos2, size_type n2) const;
int compare(size_type pos, size_type n, const Ch *p,
   size_type n2=npos) const;
```

CINECA Consorzio Interuniversitario

getline

```
#include <iostream>
#include <string>

using namespace std;

main() {
  string line;

  while(getline(cin,line)) {
    cout << line << endl;
  }
}
```

getline è definito in <string> e ritorna reference cin. Quando getline fallisce il reference a cin (operator void*) vale 0 ed il while esce.

CINECA Consorzio Interuniversitario

stringa-c++ -> c

E' spesso necessario convertire una “string” in una stringa C, solitamente per utilizzare funzioni di libreria C:

`c_str()` ritorna il puntatore ad una stringa C, terminata da 0
`data()` ritorna il puntatore ad una stringa C non terminata
`copy(char *p, size_t n, size_t pos=0)` copia la stringa in p

c_str() e data() allocano e ritornano la stringa in un buffer interno alla string. copy() permette di copiare da questo buffer temporaneo su un array di char esterno.

Esempio:

```
string s="1223";
int i=atoi(s.c_str());  // i=1223
```

CINECA Consorzio Interuniversitario

S T L

Standard Template Library
“don’t reinvent the wheel”

CINECA Consorzio Interuniversitario

Cos'è?

E' una libreria di strutture dati e algoritmi standard.

E' nata come una libreria esterna, poi è diventata parte della libreria standard C++.

Contiene moltissime funzionalità, complesse da implementare, ma che ogni programmatore "C" ha spesso utilizzato ed implementato più di una volta: liste, vettori, mappe, stringhe...

CINECA Consorzio Interuniversitario

Generica!

Si basa su classi template!

quindi applicabile a qualunque tipo di dati

può sempre venire specializzata... con l'ereditarietà!

CINECA Consorzio Interuniversitario

Performance

E’ efficiente:

- è ottimizzata nel senso che tutte le funzioni utilizzate comunemente sono inline
- non è High Performance in senso stretto (ottimizzazioni di cache, algebra lineare, FFT… usare librerie concepite appositamente).

bug-free

eseguibili "grandi‟, tempi di compilazione "lunghi": includere solo le parti necessarie, dove servono!

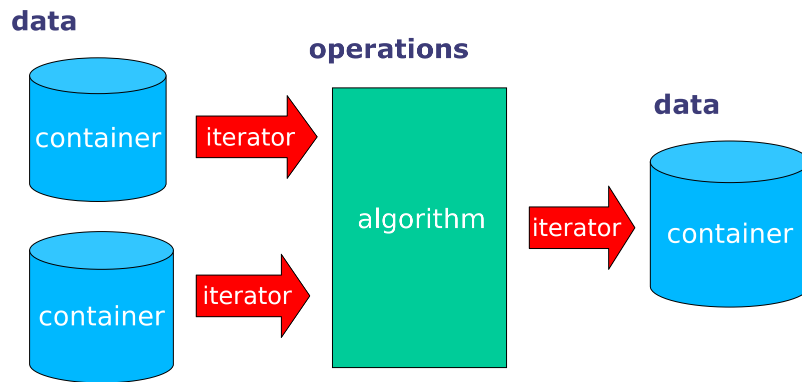
CINECA Consorzio Interuniversitario

Struttura STL

- **Containers**
  Sono oggetti STL che memorizzano dati. Oltre a fornire la memoria necessaria, definiscono anche il meccanismo di accesso ai dati che contengono (iteratori).
- **Iteratori**
  Sono una generalizzazione della nozione di puntatore: sono oggetti che puntano ad altri oggetti. Danno la possibilità di “attraversare” un container.
- **Algoritmi**
  Funzionalità che permettono di manipolare il contenuto dei container. Svolgono compiti base e più comuni.

CINECA Consorzio Interuniversitario

Funzionamento generale



Separation of data and operations.
Iterators are an interface between containers and algorithms.

CINECA Consorzio Interuniversitario

Funzionamento generale

1. Scegliere il tipo di *container* da utilizzare.
2. Per inserire, modificare, cancellare gli elementi del container usare:
   - le *funzioni membro* della classe oppure
   - gli *iteratori* del container
3. Dopo aver inserito i dati nel container, lo si può manipolare tramite *algoritmi.*

Containers

cosa possono contenere? Qualunque oggetto che:

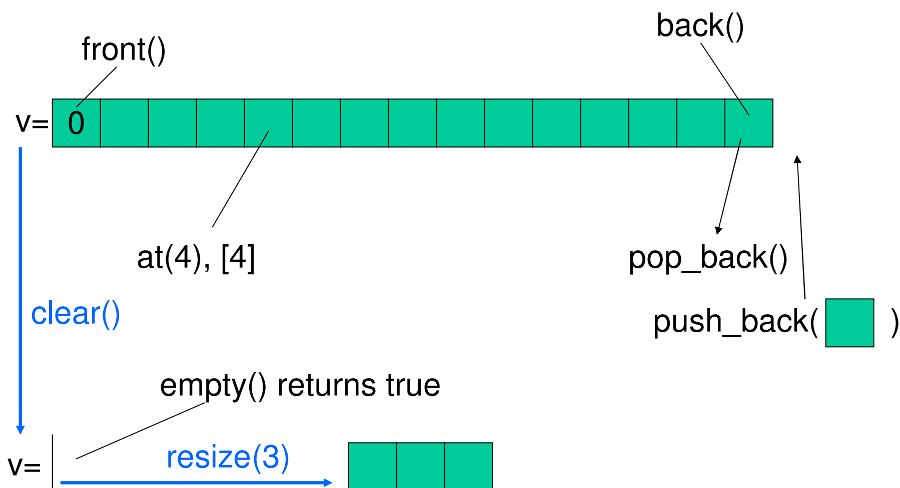
- abbia un valore di default (*default constructor*)
- abbia l'operatore di uguaglianza: == (*test of equality*)
- abbia l'operatore "minore": < (*sorting criterion*)

Sicuramente tutti i tipi built-in!

CINECA Consorzio Interuniversitario

Esempio: vector

miglioramento di un array C



CINECA Consorzio Interuniversitario

cos'ha in più vector

gestione della memoria
- automatica
- dinamica

interfaccia comune agli altri contenitori
- quindi: usabilità con gli *algoritmi generici*

nessuna perdita di efficienza rispetto alla versione "C".

CINECA Consorzio Interuniversitario

vector

```
include <vector>
using namespce std;
class mytype {
   char s[1024*1024];
...
};
main() {
  vector<int> vi;             // vi è di interi lungo 0 elementi
  vector<float> vf(1000 , 1.0);  // vf sono 1000 "uni"
  vector<mytype> vmy(1024);     // 2^30 byte = 1GB
  vector<mytype*> vmyp(1024);  // 1024 puntatori, 8kb su 64bit
                                                              //
       architecture
  vector<int> v1(10);
  vector<int> v2 = v1; // costruttore di copie: OK
  int i=vi[10];     // undefined  - NO CHECK!
  int j=vi.at(10); // lancia un'eccezione "out_of_range"
```

CINECA Consorzio Interuniversitario

E’ un vettore in senso logico. *Il punto di forza di questo contenitore è che è possibile accedere direttamente agli elementi (subscripting, o accesso casuale), senza alcun overhead.*
Ha senso anche definire vettori lunghi zero: l'allocazione dinamica è gestita dalla libreria:

```
vi.push_back(17); // alloca il primo elemento e gli assegna "17"
vi.push_back(100);    // alloca il secondo elemento e gli assegna "100"
cout << vi.size() << endl;  // "2" elementi
cout << vi.back() << endl;  // "100", l'ultimo elemento
cout << vi[vi.size()-1] << endl; // "100": vector da 0 a size-1
vi.pop_back();  // non ritorna valore: bisogna usare back() prima
vi.pop_back();  // nuovamente 0 elementi
vi.pop_back();  // UNDERFLOW - undefined (core dump?)
```

CINECA Consorzio Interuniversitario

Per controllare che non sia vuoto:
if (vi.empty()) { cout << "VUOTO\n"; }
E' possibile push e pop solo "back": il vettore inizia sempre da zero. Internamente viene fatta una riallocazione (non ad ogni push - a cura della libreria). In ogni caso l'incremento e la diminuzione della dimensione di un array sono operazioni costose: non usare vector quando questo è frequente (o usa vector di dimensione fissa). Per ridurre l'overhead si può usare reserve (capacity riporta la memoria attualmente allocata):

```
vi.reserve(10000);   // alloca 10000 elementi, anche se non modifica la size
for (i=0; i<10000; i++)
 {
   vi.push_back( f(i) );         // non ci sono nuove allocazioni.
   cout << v.size() << endl;      // "i+1"
   cout << v.capacity() << endl; // "10000" sempre
 }
```

Si può comunque superare il valore riservato, ma questo rimette in gioco il meccanismo di allocazione dinamica. Altrimenti si può gestire a mano la dimensione del vettore:

```
vi.resize(30000);      //size a 30000
vf.resize(2000, 2.0); // da 0 a 999 valgono 1.0, 1000 a 1999 2.0
                   |
                   valore con cui vengono inizializzati
                   i nuovi elementi aggiunti nel vector
```

Infine, max_size() è la massima dimensione "gestibile" dal sistema.

CINECA Consorzio Interuniversitario

Considerando che un vettore può contenere una grande quantità di dati, fare lo swap di due vettori può essere pesante. In C è prassi comune scambiare solo i puntatori. E anche vector permette la stessa cosa:

```
vector<float> v1(1000,1.1);
vector<float> v2(1000,2.2);
cout << v1[500] << " " << v2[500] << endl;  // "1.1 2.2"
v1.swap(v2);                            // scambia solo 2 puntatori
cout << v1[500] << " " << v2[500] << endl;  // "2.2 1.1"
```

`Type & vector::operator[](size_t i)` indirizzamento, subscripting
`Type & vector::at(size_t i)` indirizzamento con controllo
`void vector::push_back(Type val)` copia val at-the-end
`void vector::pop_back()` disalloca l'ultimo elemento
`Type & vector::back()` ritorna il reference dell'ultimo elemento
`size_t vector::size()` ritorna dimensione del vettore
`size_t vector::capacity()` ritorna la memoria allocata
`void vector::resize(size_t n)` esegue il resize a n elementi
`void vector::reserve(size_t n)` prealloca memoria per il vettore
`bool vector::empty()` ritorna *true* se è vuoto, *false* altrimenti
`void vector::swap(vector & v2)` scambia v1 con v2
`bool operator==(const vector & v2)` confronto lexicografico
<,>,<=,>=,!= idem

CINECA Consorzio Interuniversitario

vector iterator

iteratore ~ puntatore (soprattutto per vector!)
se abbiamo una sequenza, possiamo:

- "scorrerla", iterare su di essa;
- identificare elementi al suo interno;
- identificare elementi "successivo" e "precedente" di un elemento dato.

Il concetto, la classe, che permette di farlo è "iterator" ed è strettamente legato al contenitore: è contenuto in vector!

E' un iteratore ad accesso casuale (random access).

CINECA Consorzio Interuniversitario

vector iterator

```
vector<int> v(10,1); // v contiene 10 "uni"
vector<int>::iterator p;

p = v.begin();// p punta al primo elemento;
int i=*p;      // leggo il primo elemento in i
*p=100;        // scrivo il primo elemento: v[0] vale 100
p++;           // p punta a v[1]
p+=3;          // p punta a v[4]
p[3] = -10;    // v[7] vale -10; p punta ancora a v[4]
uguale a *(p+3) = -10
bool Q1=(p==v.begin()); // Q1 vale false
bool Q2=(p>v.begin());  // Q2 vale true
```

CINECA Consorzio Interuniversitario

scorrere un vector

operazione tipica, sempre uguale per tutti gli iterator e tutti i containers:

```
vector<string> s(10);
…
for(vector<string>::const_iterator i=s.begin(); i != s.end(); i++)
{
   cin >> *i; //errore devo usare l'iterator non-const per farlo
   cout << *i << endl; // ok, lettura permessa
}
```

CINECA Consorzio Interuniversitario

Iterators

Oggetti che permettono di scorrere un container così come si usa un puntatore per scorrere un array.

Operazioni fondamentali che definiscono il comportamento di un iteratore:

Operator *	returns the element of the actual position
Operator ++ / --	steps forward/backward to the next element
Operator == / !=	returns whether two iterators represent the same position
Operator =	assigns the position of the element to which it refers

CINECA Consorzio Interuniversitario

Iterators

Tipi base:

input (only read access and forward move)
output (only write access and forward move)
forward (I/O and forward move)
bidirectional (I/O, forward and backward move)
random (I/O and random access)

Forniscono il meccanismo per rendere gli algoritmi generici: gli algoritmi di solito hanno degli iteratori come argomenti, per cui un container deve solo fornire un modo per accedere ai suoi elementi mediante iteratori. Questo permette di scrivere algoritmi generici che operano su diversi tipi di container.

CINECA Consorzio Interuniversitario

Iteratori predefiniti

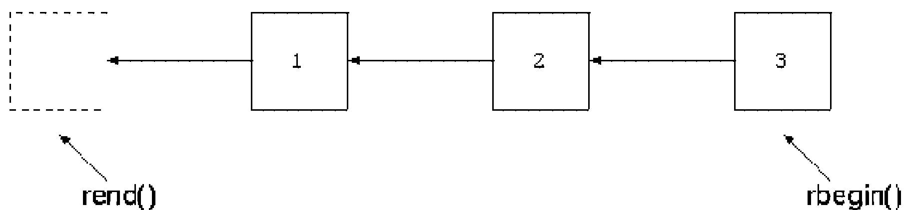
#include <iterator>

Definisce numerose classi base che forniscono il supporto per l'implementazione degli iteratori e contiene altri iteratori predefiniti, oltre ai tipi base.

- ***reverse iterators***: sono ad accesso diretto o bidirezionale e si muovono lungo la sequenza in direzione inversa.
- ***insert iterators***: permettono agli algoritmi di lavorare in modalità inserimento anziché overwrite.
- ***stream iterators***: leggono e scrivono su stream, consentono quindi agli algoritmi di manipolare degli stream.

CINECA Consorzio Interuniversitario

reverse_iterator, rbegin() e rend()



per questa ragione non si può semplicemente invertire begin ed end per scorrere una sequenza all'indietro: servono un iteratore apposito, un rbegin() e un rend()

ATTENZIONE: un reverse_iterator va indietro con "++", non con "--"!!!

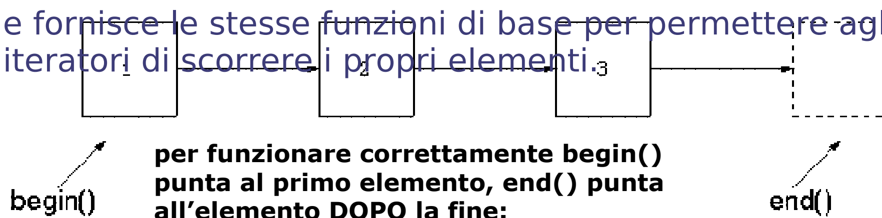
iteratori dei container

Ogni classe container definisce il proprio tipo di iteratore mediante typedef. In particolare ne definisce 2 tipi:

container::**iterator** (I/O)

container::**const_iterator** (read only)

e fornisce le stesse funzioni di base per permettere agli iteratori di scorrere i propri elementi.



begin()

per funzionare correttamente begin() punta al primo elemento, end() punta all'elemento DOPO la fine:

end()

[begin() , end() [

CINECA Consorzio Interuniversitario

Esempio

```
main() {
  vector<int> v(10);
  int k = 10;
  for (vector<int>::iterator i=v.begin(); i != v.end(); i++) {
    *i = k;
    k++;
  }
  for (vector<int>::const_iterator i=v.end(); i!=v.begin(); i--) {
    cout << *i << " " ;
  }
  cout << endl;
// "0 19 18 17 16 15 14 13 12 11" SBAGLIATO! e potrebbe dare core dump!
// comportamento indefinito
  for (vector<int>::reverse_iterator i=v.rbegin(); i != v.rend(); i++) {
    cout << *i << " " ;
  }
  cout << endl;
// "19 18 17 16 15 14 13 12 11 10"
}
```

CINECA Consorzio Interuniversitario

vector iterator methods

vector::iterator vector::begin() iteratore che punta al primo elemento

vector::iterator vector::end() one-beyond-the-end

vector::reverse_iterator vector::rbegin() iteratore che punta all'ultimo elemento (primo in ordine inverso)

vector::reverse_iterator vector::rend() one-before-the-begin

CINECA Consorzio Interuniversitario

vector iterator methods

vector::iterator vector::erase(vector::iterator p) elimina l'elemento puntato da p e ritorna ++p

vector::iterator vector::erase(first, beyond) elimina gli elementi da first (incluso) a beyond (escluso) e ritorna beyond

vector::iterator vector::insert(vector::iterator p) inserisce il default-value del Type di vector in p e ritorna p

vector::iterator vector::insert(vector::iterator p, Type val) inserisce val in p e ritorna p

void vector::insert(vector::iterator p, size_t n, Type val) inserisce n val a partire da p

vector::iterator vector::insert(p,first,beyond) inserisce la sequenza tra first e beyond a partire da p

CINECA Consorzio Interuniversitario

Esempio

```
main() {
  vector<int> v(10);
  int k = 10;
  for (vector<int>::iterator i=v.begin(); i != v.end(); i++) {
    *i = k;
    k++;
  }
  // 10 11 12 13 14 15 16 17 18 19
  v.insert(v.begin()+3, 2001);
  // 10 11 12 2001 13 14 15 16 17 18 19
  v.insert(v.end(), -100);
  // 10 11 12 2001 13 14 15 16 17 18 19 -100
  v.erase(v.begin()+1, v.end()-1);
  for (vector<int>::iterator i=v.begin(); i != v.end(); i++) {
    cout << *i << " " ;
  }
  cout << endl;
  // 10 -100
}
```

CINECA Consorzio Interuniversitario

Altre operazioni?

Perché non ci sono altre operazioni più complesse in “vector”?

Trovare uno o più elementi, ordinare un vector, sostituzioni, rimozioni condizionali, intersezione, unione,… sono tutte operazioni generali di container… infatti sono tutti ALGORITMI della STL: vedi dopo

#include <algorithm>

CINECA Consorzio Interuniversitario

Vector polimorfici

I contenitori possono contenere oggetti di un solo tipo, ma il polimorfismo permette che questo "unico tipo", siano più tipi!

Non bisogna però dimenticare la cosa essenziale: bisogna usare i puntatori! Senza si ha slicing

Segue un esempio

CINECA Consorzio Interuniversitario

```
#include <vector>
#include <iostream>
using namespace std;
class Value {
public:
  virtual void show() = 0;
};
class FLTValue : public Value {
  float v;
public:
  FLTValue() : v(0.0) {};
  FLTValue(float x) : v(x) {};
  void show() { cout << "FLT: " << v << endl; }
};
class INTValue : public Value {
  int v;
public:
  INTValue() : v(0) {};
  INTValue(int x) : v(x) {};
  void show() { cout << "INT: " << v << endl; }
};
```

CINECA Consorzio Interuniversitario

```
main() {
  vector<Value> v;

  v.push_back(FLTValue(13.7));
  v.push_back(INTValue(3));
  v.push_back(INTValue(-4));
  v.push_back(FLTValue(1000.01));

  for (vector<Value>::iterator i=v.begin(); i != v.end();
i++) {
        i->show();
  }
}
```

Questo esempio è sbagliato e non può funzionare

```
cc-1296 CC: ERROR File =
/opt/MIPSpro/MIPSpro/usr/include/CC/stl_construct.h, Line = 48
  An object of abstract class type "Value" is not allowed.
```

CINECA Consorzio Interuniversitario

```
main() {
  vector<Value*> v;

  v.push_back(new FLTValue(13.7));
  v.push_back(new INTValue(3));
  v.push_back(new INTValue(-4));
  v.push_back(new FLTValue(1000.01));

  for (vector<Value*>::iterator i=v.begin(); i != v.end(); i++) {
    (*i)->show();
  }
  while(!v.empty()) {
    delete v.back();
    v.pop_back();
  }
}
```

OUTPUT:

FLT: 13.7
INT: 3
INT: -4
FLT: 1000.01

Sequence containers

La posizione di ogni elemento dipende dal momento o dal punto di inserimento

<vector>	one dimensional array
<list>	doubly-linked list
<deque>	doubly-ended queue

CINECA Consorzio Interuniversitario

Associative containers

La posizione degli elementi dipende dal loro valore, consentono di ricercare un elemento sulla base di una chiave.

<set>	*set*: each element may occur only once *multiset*: duplicates are allowed
<map>	associative array *map*: 1 key → 1 value *multimap*: 1 key → more values
<bitset>	array of booleans

Container adapters

- Sono implementati sfruttando le classi containers fondamentali.
- Forniscono una specifica interfaccia per venire incontro a particolari esigenze;
- *Non forniscono propri iteratori.*

<stack>	LIFO stack
<queue>	*queue*: FIFO queue *priority_queue*: elements may have different priorities

CINECA Consorzio Interuniversitario

list

Una lista è una sequenza ottimizzata per l'inserimento e l'estrazione degli elementi dagli estremi

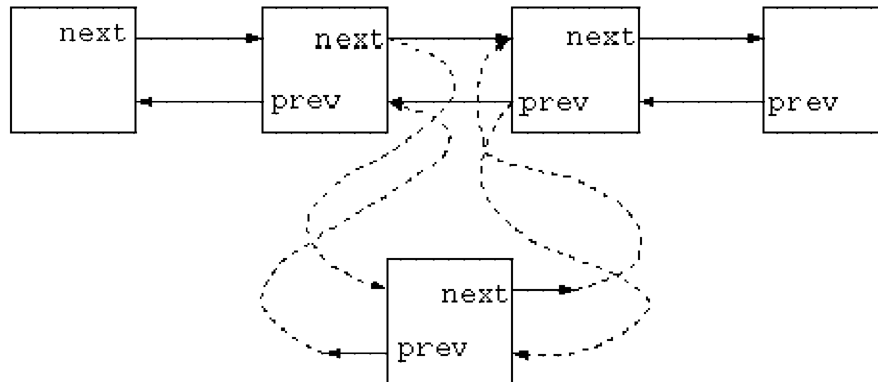
Per questo fornisce iteratori bidirezionali, ma **non ad accesso casuale**.

E' implementata solitamente come una doulby-linked list, costituita di nodi bidirezionali

CINECA Consorzio Interuniversitario

struttura

La lista contiene puntatori al primo e all'ultimo nodo. Per inserire un nuovo nodo è sufficiente allocarlo "ovunque" e quindi modificare "prev" e "next" in modo tale che questo venga puntato e punti correttamente:



CINECA Consorzio Interuniversitario

struttura(2)

L'overhead di inserimento in qualunque punto è costante e non dipende dal resto della lista!
C'è un overhead di memorizzazione: servono next e prev per ogni dato.
La velocità di scorrimento è elevata. In C:

```
node *tmp=first;
while(tmp) {
    //   do something with tmp->data
    tmp = tmp->next;
  }
```

Ovviamente in C++ gli iteratori fanno lo stesso in modo trasparente, e questo DEVE essere dimenticato!

Operazioni

list ha in comune la maggior parte delle operazioni di vector: sono entrambi sequenze.

Mancano subscripting [], at(), capacity() e reserve().

Per quando riguarda le operazioni stack-like, oltre che in coda (back), sulle liste è possibile farle anche in testa:

- push_front()
- pop_front()

CINECA Consorzio Interuniversitario

E' poi possibile unire due liste, spostando porzioni di una lista ad un'altra - splicing - senza muovere effettivamente i dati:

Importa tutta la lista x e inseriscila a partire da position:

```
void splice(iterator position, list& x)
```

Importa l'elemento puntato da i in x, dopo position:

```
void splice(iterator position, list& x, iterator i)
```

Importa gli elementi da first a last di x, a partire da position:

```
void splice(iterator position, list& x, iterator
first, iterator last)
```

Altre operazioni tipiche di spostamento dei dati, effettuate agevolmente con le liste:
Sorting della lista: `void sort()`

Per unire due liste ordinate (sorted) e ottenerne una ordinata:

```
void merge(list& x)
```

Per invertire l'ordine degli elementi della lista: `void reverse()`

CINECA Consorzio Interuniversitario

altre operazioni

Per eliminare i duplicati dalla lista:
`void unique()`
Tuttavia unique() rimuove solo quelli che compaiono consecutivamente, quindi, per rimuovere tutti i duplicati, è necessario prima eseguire il `sort()` della lista.

Per eliminare tutti i valori "value":
`void remove(const T& value)`

CINECA Consorzio Interuniversitario

Esempio

```
#include <iostream>
#include <list>
using namespace std;
main() {
  list<int> l1(3,10);
  // l1: 10 10 10
  list<int> l2;

  l2.push_back(101);
  l2.push_back(102);
  l2.push_back(103);
  // l2: 101 102 103
  l2.push_front(-1);
  l2.push_front(-2);
  l2.push_front(-3);
  // l2: -3 -2 -1 101 102 103
  // l2.begin()+3 sarebbe sbagliato!!! no random access
  l2.splice(++++++l2.begin(), l1, l1.begin());
  // l1: 10 10
  // l2: -3 -2 -1 10 101 102 103
  l2.sort();
  // l2: -3 -2 -1 10 101 102 103
```

```
  l2.merge(l1);
  // 103 102 101 10 10 10 -1 -2 -3
  l2.reverse();
  // 103 102 101 10 10 10 -1 -2 -3
  l2.unique();
  // 103 102 101 10 -1 -2 -3

  l2.remove(101);
  // 103 102 10 -1 -2 -3
  for (list<int>::const_iterator i=l1.begin(); i != l1.end();
i++) cout << *i << " " ; cout << endl;
  for (list<int>::const_iterator i=l2.begin(); i != l2.end();
i++) cout << *i << " " ; cout << endl;
}
```

CINECA Consorzio Interuniversitario

deque

DEQUE = Double Ended QUEue = list + vector
assomiglia a un vector, ma è implementata in modo tale da ottimizzare operazioni di inserimento ad entrambi gli estremi.

E' tuttavia possibile anche effettuare accesso casuale. Possiede tipi e operazioni di vector eccetto capacity() e reserve()
In più permette push_front() e pop_front(), come una lista.

Il suo utilizzo principale è quindi proprio quando si hanno principalmente operazioni agli estremi: una coda in autostrada, un mazzo di carte, ...

CINECA Consorzio Interuniversitario

stack

E' una coda LIFO: l'ultimo elemento inserito è il primo ad uscire.
Solitamente è una specializzazione di una deque, in modo da presentare un'interfaccia standard, rinominando di fatto alcune funzioni di deque, e lasciandone altre inalterate:

```
back()       -> top()
push_back()  -> push()
pop_back()   -> pop()
empty()
size()
```

Il resto di deque viene nascosto... e voilà!

CINECA Consorzio Interuniversitario

stack(2)

L'implementazione SGI parla da sola:

```
template <class T, class Sequence = deque<T> >
class stack {
  …
public:
  bool empty() const { return c.empty(); }
  size_type size() const { return c.size(); }
  reference top() { return c.back(); }
  const_reference top() const { return c.back(); }
  void push(const value_type& x) { c.push_back(x); }
  void pop() { c.pop_back(); }
};
```

CINECA Consorzio Interuniversitario

queue

Una queue è una coda FIFO: gli elementi entrano da un lato ed escono dall'altro. Anche questa normalmente è implementata con deque.

Implementa:

```
empty()
front()
back()
push()   // su back
pop()    // da front
```

CINECA Consorzio Interuniversitario

priority_queue

Analoga ad una queue (definito in <queue>), ma gli elementi con priorità più alta raggiungono la testa della coda per primi.

Fornisce:

```
empty()
size()
top()
push()
pop()
```

Il confronto tra elementi per determinarne chi ha priorità maggiore, viene eseguito con l'operator <

CINECA Consorzio Interuniversitario

map

map è un array associativo (ordinato).
Si definisce con: **map<K,V> obj;**
Un array associativo contiene coppie di valori: una key di tipo K e un valore mappato di tipo V.

map è un array associativo tradizionale, dove ogni singolo valore è associato ad una chiave unica.

map richiede che esista l'operatore "<" per il tipo della key, e gli elementi vengono memorizzati ordinati.

```
map<string,int> age;
age["Lucia"] = 25;
```

CINECA Consorzio Interuniversitario

il tipo pair

E' il tipo utilizzato per contenere le coppie di elementi. La definizione possibile:

```
template <class T1, class T2>
struct pair
{
  T1 first;
  T2 second;
…
};
```

Funzione per costruire un oggetto pair in cui il primo elemento è inizializzato a x e il secondo a y.

```
template <class T1,class T2>
pair<T1,T2> make_pair (T1 x, T2 y)
{ return ( pair<T1,T2>(x,y) ); }
```

CINECA Consorzio Interuniversitario

map iterator

Gli iteratori su map, puntano a elementi di tipo “pair”:

- first contiene la chiave
- second contiene il valore

CINECA Consorzio Interuniversitario

esempio

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

main() {
  map<string,int> age;

  age["Lucia"] = 25;
  age["Andrea"] = 23;
  age["Luca"] = 30;
  age["Lucia"] = 27;

  cout << "----------" << endl;
  cout << age[“Carlo”] << endl;
  cout << age["Luca"] << endl;

  cout << "----------" << endl;
  for (map<string,int>::const_iterator i=age.begin();
i!=age.end(); i++) {
    cout << i->first << "," << i->second << endl;
  }
}
```

OUTPUT:
0
30

Andrea,23
Carlo,0
Luca,30
Lucia,27

CINECA Consorzio Interuniversitario

metodi di map

[ ] è l'operatore di deferenziamento rispetto alla key.
In scrittura: se non esiste la chiave viene creata e il valore comunque sovrascritto con quello assegnato;
in lettura: se non esiste la chiave, viene creata e il valore viene creato con il costruttore di default.

void clear() cancella tutti gli elementi

unsigned count(key) ritorna 1 se la key è presente, 0 altrimenti

unsigned size() ritorna il n. di elementi nella map

bool empty() ritorna true se il map è vuoto, false altrimenti

void swap(map) swap di due map

iterator begin(), map::iterator end() come per vector: iteratori che puntano al primo e a one-past-the-last **rbegin(), rend()**

iterator find(key) ritorna un iteratore che punta all'elemento con chiave key, o se non esiste a end()

CINECA Consorzio Interuniversitario

metodi di map(2)

insert: permette di inserire elementi in vari modi. Il comportamento comune è che se le chiavi già esistono NON vengono sovrascritte.

pair<map::iterator, bool> insert(pair keyval) se la key non esiste già inserisce la coppia keyval, e ritorna (in un pair!) il puntatore al pair (un altro!) inserito e true. Se la key esiste già NON effettua l'inserimento, ritorna il puntatore alla key che già c'era e false.

Negli altri "insert" non viene ritornato se l'inserimento e' avvenuto o meno:
iterator insert(iterator pos, pair keyval) Idem, ma possiamo indicare anche un punto esatto o vicino a dove dovrà essere situata la coppia da inserire: può migliorare l'efficienza della ricerca.
iterator insert(iterator first, iterator beyond) inserisci gli elementi da una sequenza [first,beyond[. Gli elementi della sequenza devono puntare sempre a pair, compatibili con i tipi del map.

Esempio

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

void check_res(pair< map<string,int>::iterator , bool > res) {
 if (res.second) {
    cout << "Inserito: " << res.first->first
         << "," << res.first->second << endl;
  } else {
    cout << "Esiste gia': "<< res.first->first
         << "," <<res.first->second<<endl;
  }
}

main() {
  map<string,int> age;

  age["Lucia"] = 25; // non posso sapere se sto sovrascrivendo
                     // o creando un nuovo elemento

  pair< map<string,int>::iterator , bool > res;

  res = age.insert( make_pair("Paolo",23) ); // così invece sì
  check_res(res);

  res = age.insert( make_pair("Lucia",23) );
  check_res(res);
}
```

OUTPUT:
Inserito: Paolo,23
Esiste gia': Lucia,25

CINECA Consorzio Interuniversitario

metodi di map (3)

erase: cancella elementi dal map

- **bool erase(key)** cancella l'elemento con chiave key, se esiste, e ritorna true. Altrimenti ritorna false.
- **void erase(map::iterator pos)** cancella l'elemento puntato da pos.
- **void erase(first, beyond)** cancella gli elementi nell'intervallo di iterazione [first,beyond[

lower_bound(), upper_bound(), equal_range() esistono anche per map, ma vengono utilizzati principalmente in multimap: è li che li vediamo.

CINECA Consorzio Interuniversitario

multimap

multimap è un array associativo che permette di associare più elementi ad una stessa chiave.

L'operatore [ ] non è definito: non può più identificare univocamente un valore da una key!

CINECA Consorzio Interuniversitario

altre differenze

unsigned count(key) ritorna il n. di valori associati alla key data

unsigned erase(key) cancella tutti gli elementi con la data key

iterator find(key) ritorna il puntatore al PRIMO elemento con la data key

iterator insert(pair keyval) non ritorna un pair<iterator,bool>, ma solo iterator, visto che l'inserimento ha sempre successo!

CINECA Consorzio Interuniversitario

Esempio

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

main() {
  multimap<string,int> age;

  age.insert( make_pair("Lucia",25) );
  age.insert( make_pair("Andrea",23) );
  age.insert( make_pair("Luca",30) );
  age.insert( make_pair("Lucia",27) );

  for (map<string,int>::const_iterator i=age.begin(); i!=age.end();
i++) {
    cout << i->first << "," << i->second << endl;
  }
}
```

OUTPUT:
Andrea,23
Luca,30
Lucia,25
Lucia,27

CINECA Consorzio Interuniversitario

altri metodi

iterator lower_bound(key) ritorna il puntatore al primo elemento con la data key

iterator upper_bound(key) ritorna il puntatore “dopo” l’ultimo elemento con la data key

pair<iterator,iterator> equal_range(key) ritorna la coppia di puntatori [first,beyond[della sequenza di elementi con la data key: è meglio fare una ricerca sola piuttosto di due!

Esempio

```
#include <map>
#include <iostream>
using namespace std;
main() {
  multimap<string,int> age;
  age.insert( make_pair("Luca",30) );
  age.insert( make_pair("Lucia",25) );
  age.insert( make_pair("Andrea",23) );
  age.insert( make_pair("Luca",30) );
  age.insert( make_pair("Luca",33) );
  age.insert( make_pair("Lucia",27) );

  while(1) {
    string nome;
    cout << "Dammi il nome: ";
    cin >> nome;

    cout << "Trovati: " << age.count(nome) << endl;

    typedef map<string,int>::iterator IT;
    pair<IT,IT> seq = age.equal_range(nome);
    for (IT i=seq.first; i != seq.second; i++) {
      cout << i->second << " ";
    }
    cout << endl;
  }
}
```

OUTPUT:

Dammi il nome: Luigi

Trovati: 0

Dammi il nome: Luca

Trovati: 3

30 30 33

CINECA Consorzio Interuniversitario

set e multiset

insiemi!

set e multiset possono essere visti come map e multimap che non hanno valori associati alle key.

Gli elementi sono memorizzati in ordine (in base al loro valore), per effettuare le ricerche con tecniche binarie.

Le operazioni sono le stesse, ma non abbiamo più keyval, ma solo key:

clear(), empty(), erase(), insert(), find(), begin(),end(), lower_bound(), upper_bound(), equal_range()…

Non ci sono però alcune operazioni tipiche: unione, intersezione. Perché?

Sono definite come ALGORITMI generici a tutte le sequenze – e chiaramente ottimali per quelle ordinate!

CINECA Consorzio Interuniversitario

Algoritmi

#include <algorithm>

Gli algoritmi della libreria standard non sono funzioni membro di classi container ma funzioni globali che operano con gli iteratori.

Forniscono servizi fondamentali come *ricerca, ordinamento, copia, elaborazioni numeriche…*

CINECA Consorzio Interuniversitario

ranges

Operano su *range* di elementi, cioè su un intervallo di elementi delineato da due iteratori:

```
[begin , end[
```

Il motivo per cui i range sono definiti così, è che in tal modo gli algoritmi possono venir implementati senza dover rendere la "sequenza vuota" un caso particolare.

Alcuni algoritmi elaborano anche più di un range, ma in questo caso per i range successivi al primo è richiesto solo l'inizio perché si assume che il numero di elementi sia lo stesso del primo range.

es. `if(equal(a.begin(), a.end(), b.begin())) …`

NB. Attenzione che i range siano sempre validi!

CINECA Consorzio Interuniversitario

Esempio: copy()

`copy()` overwrites rather than inserts! So it requires that the destination has enough elements to be overwritten.

```
int main()
{
   list<int> coll1;
   vector<int> coll2;

   for(int i=1; i<=9; ++i) coll1.push_back(i);
    copy( coll1.begin(), coll1.end(),    // source
          coll2.begin());                // destination
    …
}
```

RUNTIME ERROR: overwrites nonexisting elements in the destination

CINECA Consorzio Interuniversitario

Per evitare questi errori (solo su container sequenziali):

1. garantire che la destinazione abbia abbastanza elementi (a) creandola con la dimensione giusta, oppure (b) ridimensionandola

   (a) `vector<int> coll2(coll1.size());`
   (b) `coll2.resize(coll1.size());`

oppure

2. usare *insert iterators*, essi permettono di aumentare la size della destinazione quando necessario

```
copy(coll1.begin(), coll1.end(),
     back_inserter(coll2));    //copy by appending
```

`back_inserter(`container`)`	appende con `push_back()`
`front_inserter(`container`)`	inserisce all'inizio con `push_front()`
`inserter(`container, pos`)`	inserisce da pos usando `insert()`

CINECA Consorzio Interuniversitario

argomenti funzione

Per aumentare la flessibilità e la potenza, alcuni algoritmi permettono di descrivere le operazioni da eseguire. Spesso tali funzioni sono *predicati* utilizzati per definire criteri di sort o di ricerca.

E' possibile specificare queste operazioni in due possibili modi:

1. passaggio di puntatori a funzioni definite dall'utente.
2. utilizzare *oggetti-funzione* o *funtori*: sono classi in cui il corpo della funzione è nell'overloading di **`operator()`**.

Il secondo metodo è migliore del primo perché:

- smart functions
- ogni oggetto funzione ha il proprio tipo
- migliore performance
- permette inlining
- utilizza l'incapsulazione

CINECA Consorzio Interuniversitario

Oggetti funzione

Sono oggetti che si comportano come funzioni. Cioè oggetti che si possono chiamare usando delle parentesi e passando degli argomenti.

Questo si realizza defininendo **`operator()`** con gli appropriati tipi di parametri.

```
class X{
  public:
    return-value operator() (arguments) const;
    …
};
```

Gli oggetti di questa classe si possono usare come funzioni:

```
X fo;
…
fo(arg1,arg2);
```

`fo(arg1,arg2);` è equivalente a `fo.operator()(arg1,arg2);`

Esempio

```
#include <iostream>
#include <algorithm>
#include <vector>

using namespace std;

template <class T> class Sum {
  T tot;
public:
  Sum() { tot = 0; }
  void operator() (T x) { tot += x; }
  T totale() const { return tot; }
};


main() {
  vector<float> v;
  v.push_back(12);
  v.push_back(24);
  v.push_back(17);

  Sum<float> somma;
  somma = for_each(v.begin() , v.end(), somma);
  cout << "Sommatoria: " << somma.totale() << endl;
}
```


L'algoritmo `for_each()` è scritto in questo modo:

```
template<class Iterator, class Operation>
Operation for_each(Iterator act, Iterator end,
                                         Operation op)
{
  while (act != end) {
      op(*act);
      act++;
   }
  return op;
}
```

oggetti funzione predefiniti

#include <functional>

Predicati: prendono come argomento 1 o 2 oggetti e ritornano un bool. La STL richiede che i predicati non modifichino il valore dei loro argomenti.

- binari: equal_to, not_equal_to, greater, less, greater_equal, less_equal, logical_and, logical_or
- unari: logical_not

Funzioni matematiche:

- binarie: plus, minus, multiplies, divides, modulus
- unarie: negate (cambia il segno del numero)

Vengono richiamati nella forma: **obj_funct<type>()**

CINECA Consorzio Interuniversitario

entità di supporto

1. **Binder** collegano un argomento a un oggetto funzione binario:

   bind1st(*obj-funct*, *value*)
   bind2nd(*obj-funct*, *value*)

   restituiscono un oggetto funzione unario in cui il primo (risp. secondo) operando di *obj-funct* è collegato a *value*.

2. **Negatore** restituisce il complemento di un predicato:

   not1(*predicato-unario*)
   not2(*predicato-binario*)

CINECA Consorzio Interuniversitario

esempio

```
#include <iostream>
#include <list>
#include <functional>
#include <algorithm>
using namespace std;

int main()
{
   list<int> lst;
    list<int>::iterator endp;
   for(int i=1; i<20; i++) lst.push_back(i);

// rimuovi dalla list gli elementi maggiori di 8
   endp = remove_if(lst.begin(), lst.end(),
                         bind2nd(greater<int>(),8);
   …

  return 0;
}
```

CINECA Consorzio Interuniversitario

Legenda dei tipi generici utilizzati dagli algoritmi

In, Out, For, Bi, Ran = input, output, forward, bidirectional, random access iterator

Pred = unary predicate

BinPred = binary predicate

Cmp = comparison function

Op = unary operation

BinOp = binary operation

T = tipo contenuto in sequenza

CINECA Consorzio Interuniversitario

Nonmodifying sequence operations

Op for_each(In first, In last, Op f) esegui f su ogni elemento della sequenza

In find(In first, In last, const T& val) trova la prima occorrenza di val nella sequenza
In find_if(In first, In last, Pred P) trova il primo elemento della sequenza per cui P è true.
For find_first_of(For first1,For last1, For first2, For last2) trova il primo elemento della sequenza1 che ha un match nella sequenza2.
For find_first_of(For first1, For last1, For first2, For last2, BinPred P) idem, confrontando con il predicato binario P

For adjacent_find(For first, For last) trova due elementi adiacenti uguali
For adjacent_find(For first, For last, BinPred P) idem, cfr con P

int count(In first, In last, const T& val) conta il n. di elementi che valgono val
int count_if(In first, In last, Pred P) conta il n. di elementi per cui P è true

CINECA Consorzio Interuniversitario

Nonmodifying sequence operations(2)

bool equal(In first1, In last1, In first2) true se le due sequenze sono uguali. count(seq2) >= count(seq1)
bool equal(In first1, In last1, In first2, BinPred P) idem, cfr con P

pair<In,In> mismatch(In first1, In last1, In first2) cerca la prima coppia di elementi diversi delle due sequenze e ne ritorna gli iteratori
pair<In,In> mismatch(In first1, In last1, In first2, BinPred P) idem, cfr con P

For search(For first1, For last1, For first2, For last2) cerca la sequenza2 come sottosequenza della sequenza1. Ritorna il puntatore alla seq1 dove trova la prima occorrenza, oppure last1 se non ne trova
For search(For first1, For last1, For first2, For last2, BinPred P) idem, cfr con P
For find_end(For first1, For last1, For first2, For last2) idem, ma dal fondo (backward)
For find_end(For first1, For last1, For first2, For last2, BinPred P) idem, cfr con P
For search_n(For first, For last, int n, const T& val) cerca val ripetuto n volte nella sequnza
For search_n(For first, For last, int n, const T& val, BinPred P) idem, cfr con P

CINECA Consorzio Interuniversitario

Modifying sequence operations

Out copy(In first, In last, Out res) Copia la sequenza in res
Bi copy_bacward(Bi first, Bi last, Bi res) Copia la sequenza in res in ordine invertito

Out Transform(In first, In last, Out res, Op op) applica op agli elementi della sequenza e copiali in res
Out Transform(In first1, In last1, In first2, Out res, BinOp op) idem, applicando l'operazione binaria a coppie di elementi dalle due sequenze.

For unique(For first, For last) elimina gli elementi contigui uguali dalla sequenza
For unique(For first, For last, BinPred P) elimina gli elementi contigui per cui P ritorna true
Out unique_copy(In first, In last, Out res) analogo a unique, ma non modifica la sequenza e la copia in res
Out unique_copy(In first, In last, Out res, BinPred P) idem, con il confronto P.

CINECA Consorzio Interuniversitario

Modifying sequence operations(2)

void replace(For first, For last, const T& val, const T& new_val) sostituisce new_val a val nella sequenza
void replace_if(For first, For last, Pred P, const T& new_val) sostituisce new_val se P vale true, nella sequenza
Out replace_copy(In first, In last, Out res, const T& val, const T& new_val) sostituisce new_val a val e copia la sequenza risultante in res
Out replace_copy_if(In first, In last, Out res, Pred P, const T& new_val) sostituisce new_val se P vale true e copia la sequenza risultante in res

For remove(For first, For last, const T& val) rimuove gli elementi val dalla sequenza
For remove_if(For first, For last, Pred P) rimuove gli elementi per cui P è true dalla sequenza
Out remove_copy(In first, In last, Out res, const T& val) come remove, con la sequenza risultante in res
Out remove_copy_if(In first, In last, Out res, Pred P) come remove_if, con la sequenza risultante in res

CINECA Consorzio Interuniversitario

Modifying sequence operations(3)

void fill(For first, For last, const T& val) assegna val agli elementi della sequenza
void fill_n(Out res, int n, const T& val) inserisce n elementi val nella sequenza di output res
void generate(For first, For last, Gen g) chiama ripetutamente g() per ottenere i valori da inserire nella sequenza
void generate_n(Out res, int n, Gen g) chiama n volte g() per ottenere i valori da inserire nella sequenza di output res

void reverse(Bi first, Bi last) inverti una sequenza
Out reverse_copy(Bi first, Bi last, Out res) inverti e copia
void rotate(For first, For middle, For last) ruota la seq come fosse su un cerchio, finché middle non diventa first
Out rotate_copy(For first, For middle, For last, Our res) ruota e copia
void random_shuffle(Ran first, Ran last) "mescola" la sequenza
void random_shuffle(Ran first, Ran last, Gen &g) mescola usando il generatore casuale g.
void swap(T &a, T &b) scambia 2 elementi
void iter_swap(For x, For y) scambia gli elementi puntati da 2 iteratori
For swap_ranges(For first, For last, For first2) scambia 2 intervalli

CINECA Consorzio Interuniversitario

Sorted sequences

Algoritmi di ordinamento

void sort(Ran first, Ran last) ordina la sequenza
void sort(Ran first, Ran last, Cmp cmp) ordina usando il criterio di confronto cmp
void stable_sort(Ran first, Ran last) ordina la sequenza mantenendo l'ordine degli elementi che risultano uguali
void stable_sort(Ran first, Ran last, Cmp cmp) idem usando cmp
void partial_sort(Ran first, Ran middle, Ran last) ordina la sequenza parzialmente, da first a middle.
void partial_sort(Ran first, Ran middle, Ran last, Cmp cmp) idem usando cmp
Ran partial_sort_copy(In first1, In last2, Ran first2, Ran last2) ordina la sequenza1 parzialmente, nella sequenza2. La dimensione originale della sequenza2 determina quanti elementi ordinare.
Ran partial_sort_copy(In first1, In last2, Ran first2, Ran last2, Cmp cmp) idem usando cmp
void nth_element(Ran first, Ran nth, Ran last) ordina la sequenza parzialmente, in modo che l'elemento nth non abbia elementi minori dopo di sé.
void nth_element(Ran first, Ran nth, Ran last, Cmp cmp) idem usando cmp
Bi partition(Bi first, Bi last, Pred p) metti prima tutti gli elementi che soddisfano p, poi quelli che non lo soddisfano. Ritorna il primo elemento che non lo soddisfa oppure last.
Bi stable_partition(Bi first, Bi last, Pred p) idem, ma non modificare l'ordine originale degli elementi

Sorted sequences

Algoritmi per sequenze ordinate:

bool binary_search(For first, For last, const T& val) true se val è nella sequenza
bool binary_search(For first, For last, const T& val, Cmp cmp) idem usando cmp

Per trovare gli elementi tramite ricerca binaria usiamo lower_bound/upper_bound per trovare gli estremi, oppure equal_range, per eseguire una unica ricerca invece di due, come in multimap:
For lower_bound(For first, For last, const T& val)
For lower_bound(For first, For last, const T& val, Cmp cmp)
For upper_bound(For first, For last, const T& val)
For upper_bound(For first, For last, const T& val, Cmp cmp)
pair<For,For> equal_range(For first, For last, const T& val)
pair<For,For> equal_range(For first, For last, const T& val, Cmp cmp)

A differenza del list::merge, quello generico copia gli elementi:
Out merge(In first1, In last1, In first2, In last2, Our res)
Out merge(In first1, In last1, In first2, In last2, Our res, Cmp cmp)
Out inplace_merge(Bi first, Bi middle, Bi last)
Out inplace_merge(Bi first, Bi middle, Bi last, Cmp cmp)

CINECA Consorzio Interuniversitario

set algorithms

bool includes(In first1, In last1, In first2, In last2) true se ogni elemento della seconda sequenza è anche nella prima (la seconda sequenza è contenuta nella prima, in senso “insiemistico”)
bool includes(In first1, In last1, In first2, In last2, Cmp cmp) idem usando cmp
Out set_union(In first1, In last1, In first2, In last2, Out res) (OR)
Out set_union(In first1, In last1, In first2, In last2, Out res, Cmp cmp)
Out set_intersection(In first1, In last1, In first2, In last2, Out res) (AND)
Out set_intersection(In first1, In last1, In first2, In last2, Out res, Cmp cmp)
Out set_difference(In first1, In last1, In first2, In last2, Out res) (-)
Out set_difference(In first1, In last1, In first2, In last2, Out res, Cmp cmp)
Out set_symmetric_difference(In first1, In last1, In first2, In last2, Out res) (XOR)
Out set_symmetric_difference(In first1, In last1, In first2, In last2, Out res, Cmp cmp)

CINECA Consorzio Interuniversitario

min/max algorithms

const T& min(const T& a, const T& b)
const T& min(const T& a, const T& b, Cmp cmp)
const T& max(const T& a, const T& b)
const T& max(const T& a, const T& b, Cmp cmp)

For min_element(For first, For last)
For min_element(For first, For last, Cmp cmp)
For max_element(For first, For last)
For max_element(For first, For last, Cmp cmp)

bool lexicographical_compare(In first, In last, In first2, In last2)
bool lexicographical_compare(In first, In last, In first2, In last2, Cmp cmp)

bool next_permutation(Bi first, Bi last) ritorna false se la sequenza è in ordine anti-lexicografico (e ritorna quella lexicografica)
bool next_permutation(Bi first, Bi last, Cmp cmp)
bool prev_permutation(Bi first, Bi last) ritorna false se la sequenza è in ordine lexicografico (e ritrona quella anti-lexicografica)
bool prev_permutation(Bi first, Bi last, Cmp cmp)

CINECA Consorzio Interuniversitario

Esempio

```
#include <algorithm>
#include <iostream>

using namespace std;

int main() {
  char v[] = "abc";
  cout << v << endl;
  while ( next_permutation(v,v+3) ) cout << v << endl;
}
```

OUTPUT:

abc

acb

bac

bca

cab

cba

v e v+3 sono char*, ma next_permutation() prende come argomenti due iteratori. Come può funzionare?

*I puntatori sono un tipo di iteratori built-in: su di essi possiamo operare ++, ==, * rispettivamente per incrementarli, confrontarli e dereferenziarli: trattandosi di un metodo generico funziona ed è efficiente.*

CINECA Consorzio Interuniversitario

Eccezioni di STL