



Librerie C++

Anno Accademico 2017-18

Programmazione di Sistema

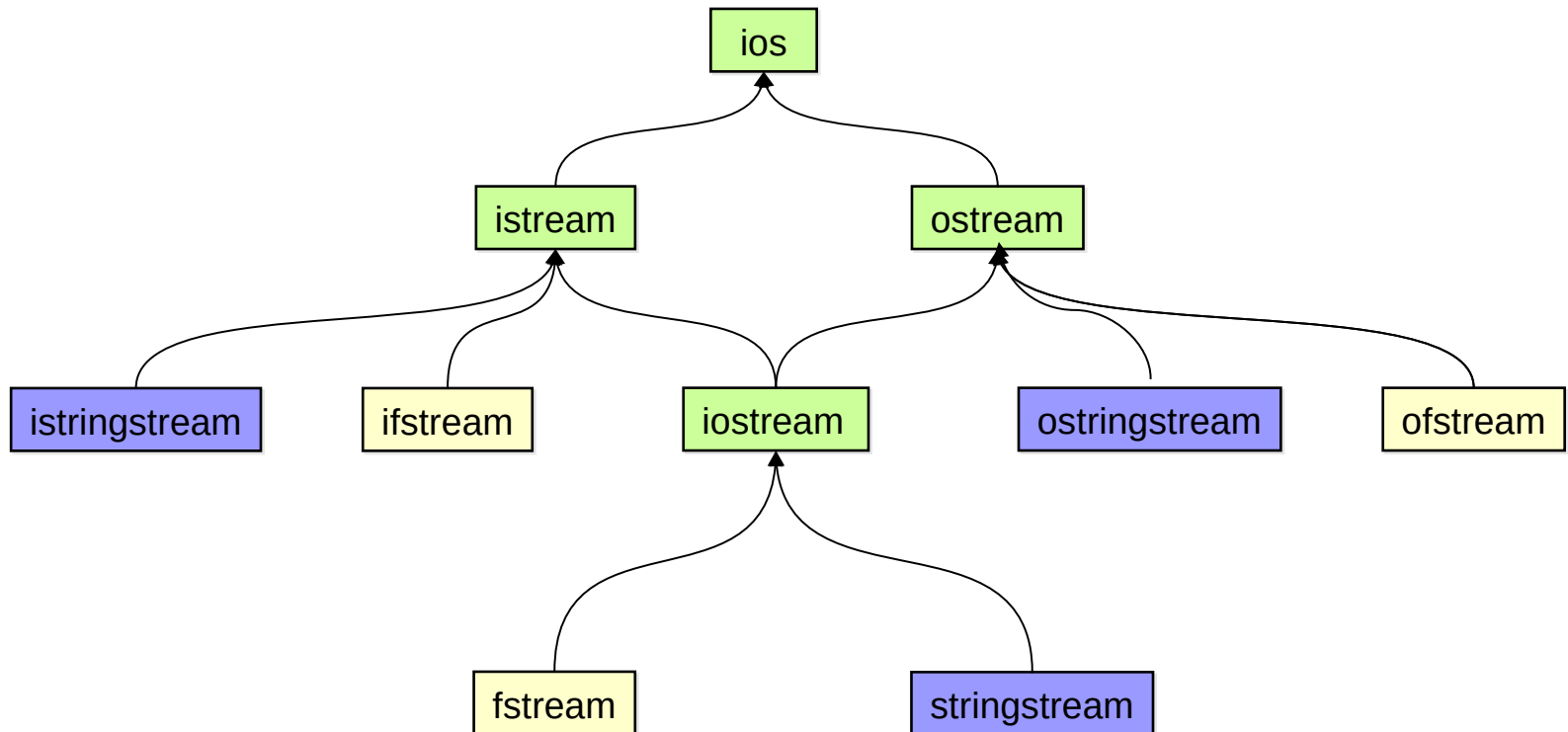


Argomenti

- Input/output in C++
- Standard Template Library

Input/Output in C++

- È possibile utilizzare le funzioni della libreria `stdio.h`
 - `printf`, `scanf`, ecc...
- Il C++ fornisce un'alternativa per la gestione dei flussi di input e di output



Gerarchia di classi di I/O (1)

- **ios** è la classe base (virtuale) e contiene attributi e metodi comuni a tutti gli stream
 - stato dello stream: (errori recuperabili, errori irrecuperabili, fine del flusso, ...)
 - manipolazione del formato
 - non può essere istanziata direttamente
- **istream** e **ostream** eseguono rispettivamente le operazioni di input ed output su flussi generici
 - metodi di lettura e scrittura dello stream
 - operatori >> e <<
 - accesso indiretto allo stream tramite buffer (allocato internamente)
- **iostream** esegue operazioni di lettura e scrittura

Gerarchia di classi di I/O (2)

- Operazioni su file
 - le classi base sono **ifstream** e **ofstream**
 - ▢ derivano (indirettamente) da **ios**
 - ▢ contengono metodi per la creazione dello stream (open, close)
 - le classi derivate definiscono stream per l'input e l'output
 - ▢ **ifstream** → accesso in sola lettura
 - ▢ **ofstream** → accesso in sola scrittura
 - ▢ **fstream** → accesso allo stream sia in lettura che in scrittura
- Esistono classi analoghe per le operazioni di lettura e scrittura su stringhe
 - **istringstream**, **ostringstream**, **stringstream**

Output su video (1)

- Per le operazioni di output
 - si utilizza la variabile `std::cout`
 - l'operatore `<<`
 - le funzioni fornite dalla classe `ostream`

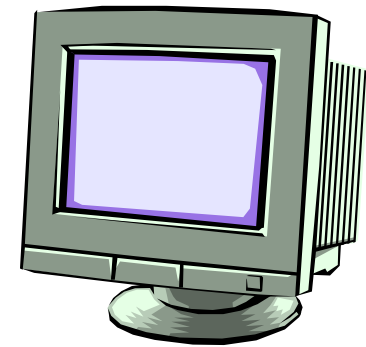


using namespace std;

<code>char ch='a';</code>	<code>cout<<ch;</code>	<code>// scrive un carattere</code>
<code>int i=10;</code>	<code>cout<<i;</code>	<code>// scrive un intero</code>
<code>float f=1.1;</code>	<code>cout<<f;</code>	<code>// scrive un float</code>
<code>char* s="ab";</code>	<code>cout<<s;</code>	<code>// scrive una stringa</code>
<code>if (!cout) { ... }</code>		<code>// Errore in scrittura</code>

Output su video (2)

- Operazioni fornite dalla classe **ostream**
 - scrittura di un carattere ch qualsiasi
 - ▮ `char ch='a'; cout. put(ch);`
 - scrittura di una blocco di byte lungo al più n
 - ▮ `cout.write(s,n);`



I/O con formato

- Funzione **width**: ampiezza del campo
 - opera sull'istruzione di scrittura successiva e definisce il numero minimo di caratteri da impiegare
 - `cout.width(4); cout<<'a';` // stampa □ □ □ a
- Funzione **precision**: precisione
 - è applicabile ai numeri reali e definisce il numero massimo di caratteri da impiegare
 - `cout.precision(4); cout<<1.4142` // stampa 1.414

Manipolatori

- Particolari oggetti, detti manipolatori, possono essere inviati in scrittura o in lettura ad un flusso
 - permettono di cambiare il formato di rappresentazione, inserire o estrarre particolari dati nel flusso
 - Definiti nel file <iomanip>

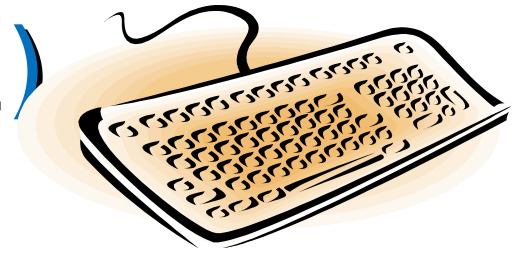
```
using namespace std;  
int i = 10;  
cout<<i<<" (0x" <<hex<<setfill('0')<<setw(4)<< i <<")"<<endl;
```

10 (0x000a)↵

Manipolatori di formato

- **skipws**: ignora gli spazi
- **left**: allinea a sinistra
- **right**: allinea a destra
- **internal**: segno a sinistra del campo, valore a destra
- **dec**: intero base decimale
- **oct**: intero base ottale
- **hex**: intero base esadecimale
- **showpoint**: reale, usa punto decimale
- **flush**: svuota il buffer in uscita
- **endl**: invia a cout il carattere '\n'
- **ends**: invia a cout il carattere '\0'
- **ws**: produce l'eliminazione degli spazi in input
- ...

Input da tastiera (1)



- Le operazioni primarie di I/O avvengono attraverso variabili standard
 - `std::cin` - flusso standard di ingresso
 - `std::cout` - flusso standard di uscita
 - `std::cerr` - flusso standard di errore (privo di buffer)
 - `std::clog` - flusso standard di errore (con buffer)
- Per le operazioni di input
 - si utilizza la variabile `std::cin`
 - l'operatore `>>`
 - le funzioni fornite dalla classe `istream`
- Utilizzando il costrutto
`using namespace std;`
si può omettere il prefisso `std::`

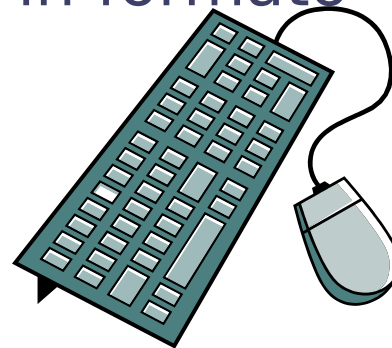
Input da tastiera (2)

- Nella lettura con l'operatore >>
 - `cin>>ch`; ignora gli eventuali spazi iniziali
 - `cin>>s`; ignora gli spazi iniziali e si arresta al primo carattere di spaziatura, la stringa viene completata con \0
 - nel caso di errore o di EOF il valore restituito da `(!cin)` è 0

```
char ch; cin>>ch; // legge un carattere
int i; cin>>i; //legge un intero
float f; cin>>f; //legge un float
char s[100]; cin>>s; //legge una stringa (fino
ad //uno spazio, tab, return,...)
if(!cin) cout << "Errore in lettura"<<endl;
```

Input da tastiera (3)

- Funzioni fornite dalla classe **istream**
 - lettura di un carattere qualsiasi
 - ▮ `cin.get(ch);`
 - lettura di una stringa di caratteri lunga al più n-1
 - ▮ `cin.getline(s,n);`
 - lettura di un blocco di dati in formato binario
 - ▮ `cin.read(s,1000);`



Gestione dello stato dello stream

- A differenza di quanto avviene in Java, in C++ le operazioni di IO non sollevano eccezioni
 - Il programmatore deve testare **esplicitamente** il risultato di ogni operazione effettuata
- Gli indicatori seguenti registrano la condizione che si è verificata a seguito dell'ultima operazione di I/O
 - **eof()** → true se è stato incontrato EOF
 - **fail()** → true se è avvenuto un errore di formato, ma che non ha causato la perdita di dati
 - **bad()** → true se c'è stato un errore che ha causato la perdita di dati
 - **good()** → true se non c'è stato alcun errore
- Il metodo **clear** ripristina lo stato del flusso

Esempio (1)



```
#include <iostream>
#include <iomanip>
using namespace std;
int main ( )
{
    int a, b, c = 8, d = 4;
    float k ;
    char name[30] ;
    cout << "Enter your name" << endl ;
    cin.getline (name, 30) ;
    cout << "Enter two integers and a float " << endl ;
    cin >> a >> b >> k ;
    if (! cin.good()) return -1; //error
}
```


Esempio (2)

```
cout << "\nThank you, " << name << ", you entered"
<< endl << a << ", " << b << ", and ";
cout.width (4) ;
cout.precision (2) ;
cout << k << endl ;
```

```
// Modo alternativo per controllare l'output
cout << "\nThank you, " << name << ", you entered"
<< endl << a << ", " << b << ", and " << setw (4)
<< setprecision (2) << k << endl ;
return 0;
}
```


Esempio - Output

Enter your name

R. J. Freuler

Enter two integers and a float

12 24 67.857

Thank you, R. J. Freuler, you entered
12, 24, and 0068

Thank you, R. J. Freuler, you entered
12, 24, and 00067.85

Esempio - I/O da file







```
#include <fstream>
using namespace std;
int main ( ) {
    int a, b, c ;
    ifstream fin ; //creazione dell'input stream
    fin.open ( "my_input.dat") ;    //apertura del file
    fin >> a >> b ;                //lettura di due interi
    if (!fin) return -1; // equivalente a if (!fin.good())...
    c = a + b ;
    ofstream fout ;                //creazione di un file di output
    fout.open ( "my_output.dat"); // apertura del file
    fout << c << endl ;            //scrittura del risultato
    fin.close ( ) ;                //chiusura input file
    fout.close ( ) ;              //chiusura output file
    return 0;
}
```

Standard Template Library

- Il comitato di standardizzazione del C++ ha definito un insieme di funzionalità, espresse attraverso template, comuni alle diverse implementazioni
 - Standard Template Library
- La libreria STL è caratterizzata da
 - classi **contenitore**, i cui oggetti sono collezioni omogenee di altri oggetti
 - vector, deque, list, set, multiset, map, multimap, **string**
 - **algoritmi** generici
 - ricerca, fusione, ordinamento,...
 - è necessario che il tipo di dato su cui viene applicato abbia definite specifiche operazioni (ad esempio ==, <, >)
- Per accedere agli elementi di una classe contenitore si utilizzano gli **iteratori**
 - oggetti di tipo cursore che si comportano come puntatori

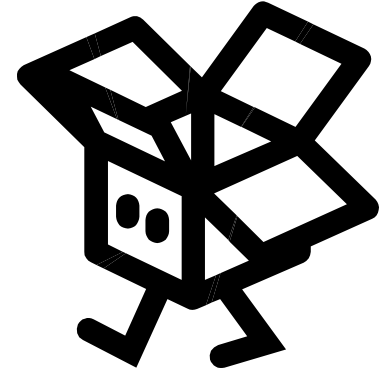
Contenitori (1)



- I contenitori di tipo **sequenziale** organizzano linearmente una collezione di oggetti
 - array “tradizionale” 
 - ▢ fornisce un accesso casuale a tempo costante ad una sequenza di lunghezza fissa
 - vector 
 - ▢ fornisce un accesso casuale a tempo costante, con tempo di inserimento e cancellazione costante in coda
 - deque 
 - ▢ fornisce un accesso casuale a tempo costante, con tempo di inserimento e cancellazione costante in testa e in coda
 - list 
 - ▢ fornisce un accesso casuale con tempo lineare e tempo di inserimento e cancellazione costante in qualsiasi posizione

Contenitori (2)

- I contenitori di tipo **associativo** forniscono l'accesso agli oggetti della collezione tramite chiavi
 - set
 - ▢ la chiave deve essere unica
 - multiset
 - ▢ la chiave può essere duplicata
 - map
 - ▢ basata su coppie chiave, valore
 - ▢ la chiave deve essere unica
 - multimap
 - ▢ la chiave può essere duplicata



Liste (1)

- Sono costituite da sequenze di dati omogenei
 - Incapsulano tutte le operazioni di gestione
 - forniscono una sintassi ed una semantica simile a quella dei tipi elementari
- Un oggetto di tipo list incapsula una lista doppiamente collegata di oggetti
 - dal punto di vista dell'utente è una sequenza di oggetti che su cui è possibile
 - ▮ inserire nuovi oggetti
 - ▮ cancellare gli oggetti
 - ▮ esaminarne il contenuto
 - si accede agli elementi di una lista tramite un iteratore



Liste (2)

- Le liste contengono dati omogenei e generici
 - ogni lista può contenere un solo tipo di oggetti
 - ▮ viene specificato alla creazione della lista
 - ▮ deve supportare copia e assegnazione
 - il compilatore impedisce che vengano inseriti dati di un tipo inappropriato
- Per utilizzare le liste è necessario includere il file `<list>`
 - definisce i template che sono usati per dichiarare e creare le liste (che appartengono allo spazio dei nomi “std”)
 - `#include <list>`
 - `using namespace std;` `//evita di dover scrivere`
 - `std::list<...>`
- Si dichiara una lista istanziandone il template
 - `list<int> w;`
 - `list<string> y;`

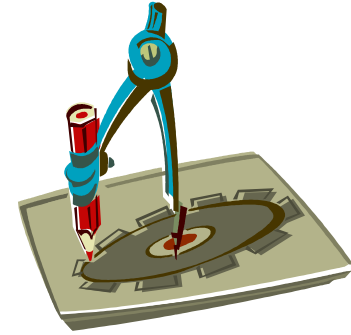


Creazione di liste (3)



- Lista vuota
 - `list <int> c0;`
- Lista con n elementi con il valore di default
 - `list <int> c1(3);` `// 0 0 0`
- Lista con n elementi del valore specificato
 - `list <int> c2(5, 2);` `// 2 2 2 2 2`
- Utilizzando l'allocatore di un'altra lista
 - `list <int> c3(3, 1, c2.get_allocator());` `// 1 1 1`
- Copiando un'altra lista
 - `list <int> c4(c2);` `// 2 2 2 2 2`

Iteratori



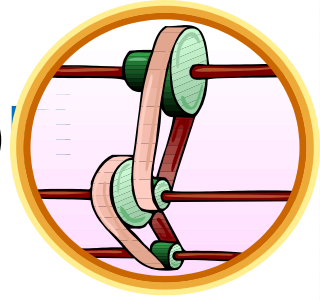
- Oggetti che indicano una posizione all'interno di un contenitore
 - Si accede all'oggetto dereferenziando l'iteratore (come fosse un puntatore)
- Se si incrementa un iteratore (`operator++()`), si avanza all'elemento successivo
 - Viceversa, si torna all'elemento precedente se lo si decrementa (`operator--()`)
- Due iteratori possono essere confrontati per eguaglianza (`operator==(())`) o diversità (`operator!=(())`)

Puntatori come iteratori

- L'iteratore più semplice è un puntatore ad un blocco di oggetti
- Si delimita il blocco con due puntatori
 - L'inizio (incluso)
 - La fine (esclusa)
- In un contenitore vuoto, inizio e fine coincidono
- Dereferenziare il puntatore finale è un errore
 - Si punterebbe oltre il limite dell'array

```
class C;  
C array[10];  
  
C *iter = array;  
C *end = array+10;  
  
for( ; iter != end; iter+  
+) {  
    C elem =  
    *iter;  
    //...  
}
```

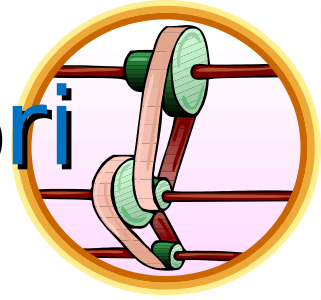
Generalizzare gli iteratori



- Il codice precedente può essere generalizzato
- Invece che operare su puntatori, può operare su oggetti di altro tipo purché possano essere
 - Confrontati (operator==, operator!=)
 - Incrementati (operator++)
 - Dereferenziati (operator*, operator->)

```
class C;  
list<C> l(10);  
  
list<C>::iterator iter =  
    l.begin();  
list<C>::iterator end =  
    l.end();  
  
for( ; iter != end; iter++) {  
    C elem = *iter;  
    //...  
}
```

Generalizzare gli iteratori



- Il codice dell'algoritmo non è cambiato
 - È cambiato il tipo di dato usato per scorrere il contenitore
- Il compilatore si occupa di selezionare le corrette operazioni (definite nella classe dell'iteratore) per eseguire confronti, incrementi e accesso al contenuto
 - Ogni contenitore offre un proprio tipo di iteratore in grado di implementare la necessaria semantica

Liste - Operazioni (1)

- **push_back** aggiunge un elemento alla fine della lista
- **push_front** aggiunge un elemento all'inizio della lista
- **back** restituisce il riferimento all'ultimo elemento della lista
- **front** restituisce il riferimento al primo elemento della lista

```
list<int> c1;  
c1.push_back( 10 ); c1.push_back(11);  
int& i = c1.back( );  
int& j = c1.front( );
```

Liste - Operazioni (2)

- **erase** rimuove un elemento o una serie di elementi a partire dalla posizione specificata
`c1.push_back(10);`
`c1.push_back(20);`
`c1.erase(c1.begin());`
- **insert** inserisce uno o più elementi nella lista nella posizione specificata
`list <int>::iterator lter;`
`c1.push_back(10);`
`c1.push_back(20);`
`lter = c1.begin();`
`lter++;`
`c1.insert(lter, 100);`

Liste - Operazioni (3)

- **clear** elimina tutti gli elementi della lista
`c1.clear();`
- **empty** verifica se la lista è vuota
`bool b = c1.empty();`
- **size** restituisce il numero di elementi nella lista
`list <int>::size_type l = c1.size();`
- **max_size** restituisce il numero massimo di elementi che la lista può contenere
`list <int>::size_type m = c1.max_size();`
- **pop_back** cancella l'ultimo elemento della lista
`c1.pop_back();`
- **pop_front** cancella il primo elemento della lista
`c1.pop_front()`

Liste - Operazioni (3)

- **remove_if** cancella tutti gli elementi per cui è verificata la condizione specificata

```
class is_odd :  
    public std::unary_function<int, bool> {  
        public: bool operator( ) ( int val ) {  
            return ( val % 2 ) == 1; }  
    };  
  
int main( ) {  
    list<int> c1;  
    c1.push_back( 3 ); c1.push_back( 4 ); c1.push_back( 5 );  
    c1.push_back( 6 ); c1.push_back( 7 ); c1.push_back( 8 );  
    c1.remove_if( is_odd( ) );    // c1 = 4 6 8  
}
```


Liste - Operazioni (4)

- **merge** elimina gli elementi della lista passata come argomento e li inserisce nella lista destinazione, riordinandola in ordine crescente (o nell'ordine specificato)

```
list <int> c1, c2, c3;  
list <int>::iterator c1_iter, c2_iter, c3_iter;  
c1.push_back( 3 ); c1.push_back( 6 );  
c2.push_back( 2 ); c2.push_back( 4 );  
c3.push_back( 5 ); c3.push_back( 1 );  
c2.merge( c1 ); // 2 3 4 6  
c2.merge( c3, greater<int>( ) ); // 6 5 4 3 2 1
```

Liste - Operazioni (5)

- **remove** cancella l'elemento passato come parametro
`c1.push_back(5);`
`c1.remove(5);`
- **resize** specifica la nuova dimensione della lista
`c1.push_back(10); c1.push_back(20);`
`c1.push_back(30);`
`c1.resize(6,40); // c1 =`
`10,20,30,40,40,40`

Liste - Operazioni (5)

- **reverse** inverte l'ordine degli elementi
`c1.reverse();`
- **sort** ordina gli elementi in ordine ascendente o secondo l'ordinamento specificato
`c1.push_back(20);`
`c1.push_back(10);`
`c1.push_back(30);`
`c1.sort();` `// c1 = 10 20 30`
`c1.sort(greater<int>());` `// c1 = 30 20 10`

Liste - Operazioni (6)

- **splice** cancella gli elementi dalla lista passata come argomento e li inserisce nella lista destinazione

```
list<int> c1, c2;  
c1.push_back( 10 ); c1.push_back( 11 );  
c2.push_back( 12 ); c2.push_back( 20 );  
c2.push_back( 21 );  
  
c2.splice(c2.begin( ), c1);  
// c2 = 10 11 12 20 21
```

Liste - Operazioni (7)

- **unique** rimuove gli elementi adiacenti duplicati

```
c1.push_back( -10 );  
c1.push_back( 10 );  
c1.push_back( 10 );  
c1.push_back( 20 );  
c1.push_back( -10 );  
c1.push_back( 20 );  
c1.sort();  
//c1 = -10-10 10 10 20 20  
c1.unique( );  
// c1 = -10 10 20
```

Liste - Operazioni (7)

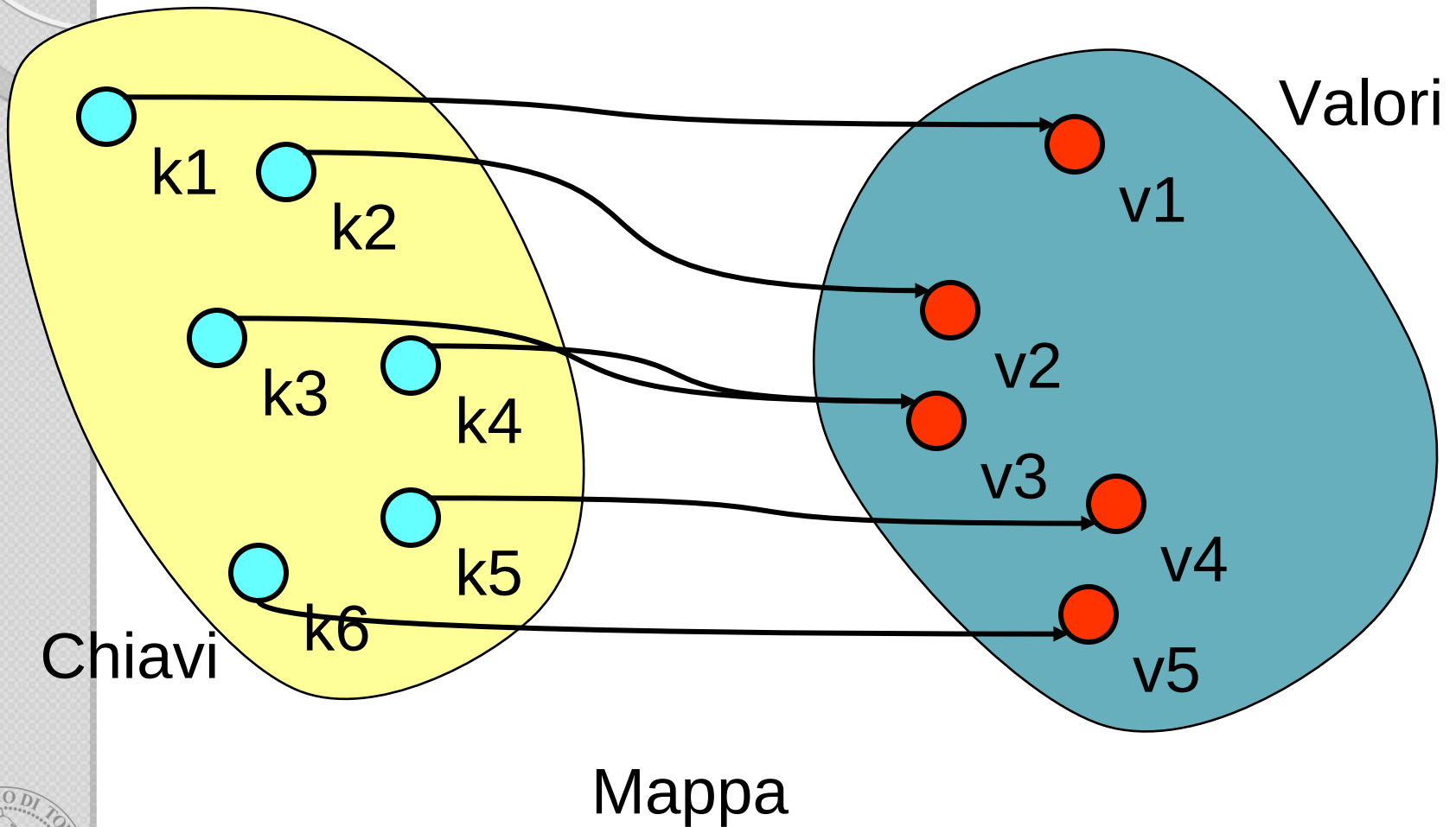
- **swap** scambia gli elementi di due liste

```
c1.push_back( 1 );  
c1.push_back( 2 );  
c1.push_back( 3 );  
c2.push_back( 10 );  
c2.push_back( 20 );  
c3.push_back( 100 );  
c1.swap( c2 );    // c1 = 10 20  
swap( c1,c3 );    // c1 = 100
```

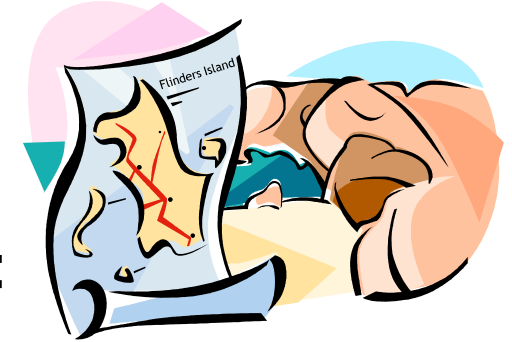
Mappe (1)

- Modellano il concetto di relazione tra due insiemi di oggetti, detti rispettivamente chiavi e valori
 - Una mappa associa, ad ogni chiave, uno ed un solo valore
 - Le chiavi devono essere oggetti immutabili
- Per utilizzare le mappe è necessario includere il file `<map>`
 - `#include <map>`

Mappe (2)

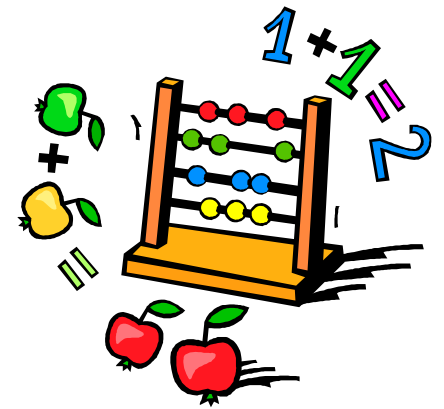


Mappe (3)



- Una mappa è un contenitore:
 - di tipo *sorted associative*
 - ▮ mantiene gli oggetti ordinati in base alle loro chiavi
 - di tipo contenitore di tipo *pair associative*
 - ▮ `pair <const Key, Data>`
 - di tipo *unique associative*
 - ▮ due elementi non possono avere la stessa chiave
- Per accedere agli elementi di una mappa, si utilizzano gli iteratori
 - l'inserimento di un elemento non invalida gli iteratori esistenti che puntano a elementi esistenti
 - la cancellazione di un elemento non invalida gli iteratori, tranne quelli che puntano all'elemento cancellato

pair <t1,t2>



- Coppia di elementi eterogenei
- t1 e t2 devono essere modelli di tipo Assignable
 - un tipo di dato è Assignable se
 - ▮ è possibile costruire una copia delle sue istanze
 - ▮ è possibile assegnare un nuovo valore ad una variabile che contiene un'istanza precedente
- Per accedere al primo elemento della coppia
 - `pair.first;`
- Per accedere al secondo elemento della coppia
 - `pair.second;`

Operatori (1)

- **insert** – inserisce uno o più elementi in una mappa

```
map <int, int> m1;  
typedef pair <int, int> Int_Pair;  
m1.insert ( Int_Pair ( 0, 0 ) );  
m1.insert ( Int_Pair ( 1, 1 ) );  
m1.insert ( Int_Pair ( 2, 4 ) );
```

Operatori (2)

- **begin** – restituisce l'iteratore alla posizione del primo elemento della mappa

```
map <int, int> m1;  
map <int, int> :: iterator m1_iter;  
typedef pair <int, int> Int_Pair;  
m1.insert ( Int_Pair ( 0, 0 ) );  
m1.insert ( Int_Pair ( 1, 1 ) );  
m1.insert ( Int_Pair ( 2, 4 ) );  
m1_iter = m1.begin ( );
```

Operatori (3)

- **end** – restituisce un iteratore alla posizione successiva all'ultimo elemento della mappa

```
map <int, int> m1;  
map <int, int> :: iterator m1_iter;  
typedef pair <int, int> Int_Pair;  
m1.insert ( Int_Pair ( 1, 10 ) );  
m1.insert ( Int_Pair ( 2, 20 ) );  
m1.insert ( Int_Pair ( 3, 30 ) );  
m1_clter = m1.end( );
```

Operatori (4)

- **empty** – restituisce true se la mappa è vuota
- **clear** – cancella tutti gli elementi della mappa
- **count** – restituisce il numero di elementi la cui chiave è quella passata come parametro

```
map<int, int> m;  
map<int, int>::size_type i;  
typedef pair<int, int> Int_Pair;  
m1.insert(Int_Pair(3, 4));  
i = m.count(3); // i = 1
```

Operatori (5)

- **equal_range** – restituisce una coppia di iteratori che identificano l'intervallo di elementi la cui chiave è pari al valore specificato

```
typedef map <int, int, less<int> > IntMap;  
typedef pair <int, int> Int_Pair;  
m1.insert ( Int_Pair ( 1, 10 ) );  
m1.insert ( Int_Pair ( 2, 20 ) );  
m1.insert ( Int_Pair ( 3, 30 ) );  
pair <IntMap::const_iterator, IntMap::const_iterator> p1,  
    p2;  
p1 = m1.equal_range( 2 );  
p1.first -> second;           // vale 20  
p1.second -> second;         // vale 30
```


Operatori (6)

- **erase** – rimuove uno o piu' elementi dalla posizione specificata
`Iter1 = ++m1.begin();`
`m1.erase(Iter1);`
- **max_size** – restituisce la dimensione massima della mappa
- **size** – restituisce il numero di elementi della mappa

Operatori (7)

- **find** – restituisce un iteratore che punta all'elemento la cui chiave è quella passata come parametro

```
m1.insert ( Int_Pair ( 1, 10 ) );
```

```
m1.insert ( Int_Pair ( 2, 20 ) );
```

```
m1.insert ( Int_Pair ( 3, 30 ) );
```

```
map <int, int> :: const_iterator m1Iter =  
    m1.find( 2 );
```

Operatori (8)

- **operator[]** – inserisce o modifica il valore di un elemento della mappa
`m1[1] = 10;`
- **get_allocator** – restituisce una copia dell'allocatore utilizzato per costruire la mappa

```
map <int, int>::allocator_type m1_Alloc;  
map <int, int, allocator<int> > m1;  
m1_Alloc = m1.get_allocator( );
```

Operatori (9)

- **lower_bound** – restituisce un iteratore al primo elemento il cui valore della chiave è maggiore o uguale di quello specificato
- **upper_bound** – restituisce un iteratore al primo elemento il cui valore della chiave è maggiore di quello specificato
- **swap** – scambia gli elementi di due mappe

Esempio (1)

```
struct ltstr {  
    bool operator()(const char* s1, const char* s2) const  
    {  
        return strcmp(s1, s2) < 0;  
    } };  
  
int main() {  
    map<const char*, int, ltstr> months;  
    months["january"] = 31;  
    months["february"] = 28;  
    months["march"] = 31;  
    months["april"] = 30;  
    months["may"] = 31;  
    months["june"] = 30;  
    months["july"] = 31;  
    months["august"] = 31;  
  
    //... continua
```



Esempio (2)

```
months["september"] = 30;  
months["october"] = 31;  
months["november"] = 30;  
months["december"] = 31;
```

```
cout << "june -> " << months["june"] << endl;
```

```
map<const char*, int, ltstr>::iterator cur = months.find("june");  
map<const char*, int, ltstr>::iterator prev = cur;  
map<const char*, int, ltstr>::iterator next = cur;  
++next;  
--prev;  
cout << "Previous (in alphabetical order) is " << (*prev).first << endl;  
cout << "Next (in alphabetical order) is " << (*next).first << endl;  
}
```