

Standard template library

- Le librerie preconfezionate permettono di semplificare molto lo sviluppo di programmi
- In questo modo però, la possibilità di sviluppare un programma dipende dalla disponibilità di quelle librerie sul computer che si sta usando
- Sarebbe auspicabile, per aumentare la portabilità del codice, avere le stesse librerie dappertutto
- La soluzione è avere delle librerie ottimizzate all'interno del linguaggio C++
- È ciò che si è fatto con la Standard Template Library

Template

- Voglio che una stessa funzione operi su dati di tipo diverso
- Il meccanismo usato fin qui è l'overloading delle funzioni
- Ogni funzione deve essere scritta una volta per ogni tipo di dato su cui agisce, con perdita di tempo e possibilità di errore
- La soluzione al problema sono i template di funzione

Template di Funzione

- È definito da

```
template<typename T> fun(typename T x)
```

- All'interno della funzione x rappresenta l'istanza di un oggetto di tipo T
- La funzione può essere poi chiamata su oggetti di tipo diverso
- Esempio

```
template<typename T> void fun(typename T x)
{
    cout << x << endl;
}
int main()
{
    double a = 2.71828;
    int b = 12;

    fun(a);
    fun(b);

    return 0
}
```

Funzioni template con argomento classe

- L'argomento del template può essere una classe

```
template<class T> double fun(class T x)
```

- In questo caso occorre fare attenzione alle istruzioni all'interno delle funzione
- Se utilizzo "=". ad esempio, bisogna che l'assegnamento sia ridefinito per i membri della classe T
- Esempio:

```
template<class T> void myCopy(class T x, class T y)
{
    x = y;
}
```

Classi template

- Si possono definire classi template

```
template<class T> class Vettore
{
    private:
        int n;
        T *v;
    public:
        ...funzioni varie..
}
```

- Anche qui fare attenzione alle operazioni usate nelle funzioni, che devono essere definite per quella classe
- Un'istanza di questa classe, ad esempio per numeri in doppia precisione, si crea con l'istruzione

```
Vettore<double> a;
```

Separazioni

- La STL è progettata in modo da tenere ben separate le strutture in cui si memorizzano i dati dalle funzioni che agiscono su di essi
- Le strutture, chiamate contenitori, sono di diverso tipo e con diverse proprietà di accesso
- Alcune funzioni agiscono su tutti i contenitori, altre solo su alcuni
- L'unione tra contenitori e funzioni è realizzata dagli iteratori

Contenitori

- Ne esistono diversi tipi, ma mi limito a occuparmi di questi
 - vector
 - list
 - stack
 - queue
 - valarray
- vector e valarray sono sicuramente i più utili per noi, perché permettono un accesso a tutti gli elementi come se fossero array
- list è ottimizzata per inserire elementi in qualunque posizione
- stack inserisce e toglie elementi dalla testa
- queue inserisce elementi in testa e li elimina in coda

Iteratori

- Sono simili, per molti aspetti, ai puntatori, ma non bisogna trattarli allo stesso modo; per esempio, non si può confrontare in iteratore con NULL
- Si possono dereferenziare ($*p$ ha senso, se p è un iteratore)
- Si possono incrementare e decrementare ($p++$, $p--$, $p+=3$, etc.)
- Si possono confrontare ($p_1 < p_2$ se p_2 punta a un elemento successivo)

Funzioni contenute nella classe del contenitore

Alcune funzioni sono disponibili per tutti i contenitori

- **begin()**
restituisce un iteratore che punta al primo elemento
- **end()**
restituisce un iteratore che punta all'elemento dopo l'ultimo
- **size()**
restituisce il numero di elementi nel contenitore
- **clear()**
rimuove tutti gli elementi dal contenitore
- **empty()**
ritorna il valore "true" se il contenitore è vuoto
- **erase(it)**
*cancella l'elemento *it* di un vector*
- **erase(it1, it2)**
*cancella gli elementi da *it1* a *it2-1* di un vector*
- **push_back(x)**
*aggiunge un elemento di valore *x* in coda al vettore, occupandosi di allocare la memoria necessaria. L'analogo per uno stack è *push**

- `pop_back()`
rimuove un elemento da un vettore; l'analogo per uno stack è `pop`
- `insert(it,x)`
inserisce in un vettore l'elemento `x` al posto puntato da `it`
- `top()`
accede all'elemento in cima allo stack

Funzioni esterne alla classe

Per utilizzare queste funzioni è necessario includere il file d'intestazione `<algorithm>`

Notazione: `it1, it2, ot1` iteratori; `v` valore, `f` funzione

- `for_each(it1,it2,f)`

Esegue la funzione f su tutti gli oggetti puntati da it1 compreso fino a it2 escluso. Sostituisce un ciclo

- `find(it1, it2, v)`

Trova la prima occorrenza di valore v in una sequenza e restituisce un iteratore all'elemento trovato (o a quello dopo l'ultimo, se non ce ne sono)

- `count(it1, it2, v)`

Restituisce il numero di elementi uguali a un certo valore v

- `count_if(it1, it2, f)`

Restituisce il numero di elementi per cui la funzione booleana restituisce "true"

- `fill(it1,it2,v)`

Assegna valore v a tutti gli elementi che hanno iteratore compreso tra it1 e it2-1

- `fill_n(it1, Size n, v)`

Assegna il valore v ad n elementi a partire da quello puntato da it1

- `generate(it1,it2,f)`

Riempie i valori tra quelli puntati da it1 a it2-1 con il risultato della funzione f

- `generate_n(it1,n,f)`

Lo stesso ma per n valori a partire da quello puntato da it1

- `transform(it1,it2,ot1, f)`

Applica la funzione f ad ogni elemento tra quelli puntati da it1 e it2-1 e mette il risultato nello stesso numero di elementi a partire da ot1. Può essere anche ot1=it1

- `sort(it1,it2)` *Ordina gli elementi*

- `sort(it1,it2, f)` *Ordina gli elementi tramite la funzione f(i,j), che ritorna "true" se l'elemento i precede l'elemento j, falso altrimenti*

- `min_element(it1,it2)` e `min_element(it1,it2,f)`
Ritornano un iteratore che punta al primo elemento della sequenza: nel primo caso è quello minore di tutti, nel secondo è il primo in base alla regola detta dalla funzione f

- `max_element(it1,it2)` e `max_element(it1,it2,f)`
Analogo al precedente, ma per il massimo anziché per il minimo

Valarray

- È una variante dei vettori pensata per i calcoli scientifici;
- è meno versatile di *vector*, perché l'idea è di ottimizzare per la velocità;
- consente di applicare una funzione a tutto il vettore in un'unica istruzione;
- consente di estrarre un sottoinsieme usando le *slice*.

Funzioni disponibili con valarray

- *Funzioni matematiche: se v è dichiarato come*

```
valarray<double> v;
```

allora

```
sqrt(v);
```

è un valarray le cui componenti sono ciascuna la radice quadrata della componente corrispondente di v . Lo stesso vale per le funzioni trigonometriche, iperboliche, esponenziali e logaritmiche;

- *un valarray può essere assegnato in blocco ad un altro: $v_1 = v_2$;*
- *un valarray può essere creato con un certo numero di elementi, come copia di un altro, oppure riempito di valori costanti, con i comandi:*

```
valarray<double> v1(100); // valarray di 100 elementi
valarray<int> v1(v2);    // v1 e' una copia di v2
valarray v1(50,3.14);    // contiene 50 elementi
                        // ciascuno vale 3.14
```

- *somma, sottrazione, $+=$, $-=$, $*=$, $/=$, sono tutti già definiti per agire su tutto l'array.*

Funzioni membro

Alcune funzioni sono definite come membri di `valarray`

- *sum()* somma gli elementi dell'array;
- *min()* restituisce l'elemento minimo;
- *max()* restituisce l'elemento massimo;
- *apply(f)* rimpiazza tutti gli elementi v_i con $f(v_i)$;
- *size()* restituisce il numero di elementi;
- *resize(n, K)* cambia ad n il numero di elementi e il valore di ciascuno sarà K .

Slice

Posso selezionare elementi di un valarray e memorizzarli in un altro valarray usando slice

```
// creo e riempio un valarray di 1000 elementi
vararray<double> v1(1000, 6.28);
// uso slice per riempire un nuovo valarray
valarray<double> v2 = v1(slice(3,100,4));
```

il secondo comando seleziona **100** elementi di **v1** a partire da quello con indice **3** e saltando poi di **4** posti al successivo fino ad avere ottenuto **100** elementi. in pratica verranno selezionati $v_1[3], v_1[7], v_1[11], \dots, v_1[299]$ e verranno posti in $v_2[0], \dots, v_2[99]$

Se ho una matrice **100x100** memorizzata in un vettore **M**, e voglio selezionare la riga con indice **k** e metterla in un vettore **v**, posso scrivere

```
valarray<double> v = M(slice(100*k,100,1))
```

Se invece voglio selezionare la riga con indice **k** scriverò

```
valarray<double> v = M(slice(k,100,100))
```

Altre funzioni

- Ci sono molte altre funzioni nella libreria standard
- Sono quasi un centinaio, molte sovraccaricate
- La documentazione si può trovare sul sito www.cplusplus.com
- È prevedibile che, nel futuro, i programmi verranno scritti usando questa libreria il più possibile