Template

Questo non è un riassunto "buono", ma sono annotazioni, quindi usatelo solo per ripasso dei concetti cardine

Stessa funzione ma con parametri diversi

TEMPLATE: permette di avere il tipo della variabile parametrico, in modo da adattare lo stesso codice a più tipi.

S.T.L: standard template library = librearia basata sui template.

```
void swap(int& x, int& y){
          int tmp=x; x=y; y=tmp; }
```

Diventa come template

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

Il linker a compile time sostituisce il template con il tipo attuale. (Polimorfismo a compile time)

```
int i,j;
swap(i,j); // Use swap with T replaced by int
String s,t;
swap(s,t); // Use swap with T replaced by String
```

Utilizzando i template almeno un parametro deve essere di tipo template

```
template<class T, class S> T f(T &, S &); //OK
template<class T, class S> T f(S &); //ERROR
```

In alcune circostanze è necessario fare l'overload degli operatori per poter usare efficientemente i template.

Nel caso di swap l'operatore = dovrà subire overloading in base al tipo del parametro (es. uguagliaza tra stringa e num diverse)

Esempio di classe

```
template <class T> class Complex {
private:
T re,im;
public:
```

```
Complex (const T& r, const T& i):re(r),im(i){}
T getRe() {return re;}
T getIm() {return im;}
}
Complex <double> x(1.0,2.0) // T = double
Complex <int> j(3,4) // T = int
Complex <char*> str(“1.0”,“6”) // T = char *
```

Vettore dinamico

```
template <class T, int dim> class Message{
private:
T mess[dim];
....
```

STL Standard template library

Scritta Alex Stepanov in 1976.

Il nucleo della STL è composto da container, algoritmi, iteratori.

- Container
  Oggetti che contengono altri oggetti.
  Lists -> vector, list, deque
  Adaptors -> queue, priority_queue, stack
  Associative -> map, multimap, set, multiset
- Algoritmi
  Operano sui container (ordinamento, ricerca...)
- Iteratori
  Attraversano il contenuto di un container

Uso dei vector

```
vector<int> A;
push_back(const T &val); // Add a new element at the end of the vector
pop_back() // Remove one element at the end of the vector
size_type size() // Returns the number of elements in the vector
bool empty() // Returns true if there are no elements in the vector
```

```
otherwise, it returns false
void resize(size_type s) // The number of elements in the vector is now s.
void clear() //Removes all elements
//What if we want to retrieve or change one element?
A.at(0) = A.at(1) + 1;
A = B Assignment
Logical equality operators == and != work too
if (B == A) { // same size, same elements?

int find (const vector<T> &vect, T target);
  Loops through the elements in the vector, searching for an
  element equal to target
  Returns index of target if it's found.
  If not found, return either -1 or vect.size()

void deleteAt (vector<T> &vect, int idx);
//Remove the element at index idx (if it exists)
void insertAfter (vector<T> &vect, T newItem, int idx);
//Add newItem after element with index idx
```

Iterators

v.begin() and v.end() return iterators

```
for (vector<int>::iterator i =
  v.begin(); i != v.end(); ++i)
  cout << *i;
```

list<T>

Bidirectional, linear list, sequential access only

Constructors

```
  list<T>() |  list<T>(size_t num_elements) |  list<T>(size_t num, T init)
```

Properties

```
  l.empty() // true if l has 0 elements
  l.size() // number of elements
//Adding/deleting elements
  l.push_back(43);
```

```
  l.push_front(31);
  l.insert(iterator,4) // insert 4 before the position “iterator”
  etc..
//Accessing elements
  l.front() // T &
  l.back() // T &
  l.begin() // list<T>::iterator
  l.end() // list<T>::iterator
// Removing elements
 l.pop_back() // returns nothing
 l.pop_front() // returns nothing
 l.erase(iterator i)
 l.erase(iter start, iter end) // delete a range
// Other operations
 l.sort(), l.sort(CompFn) // sorts in place
 l.splice(iter b, list<T>& grab_from)
```

Hashtables / Map

“associative container”

Mappa(Key, Value)

```
m.insert(make_pair(key, value)); // inserts
m.count(key); // times occurs (0, 1)
m.erase(key); // removes it
m[key] = value; // inserts it into the table
m[key] //retrieves or creates a “default” for it
i=m.begin(), i=m.end() // iterators
i->first, i->second // per accedere a chiave e
valore della coppia puntata da i
```

Hash_{…}

hash_map | hash_multimap | hash_set | hash_multiset

Iterators

- An iterator is like a pointer
- You can increment to it to go to the “next” element

- Most useful when combined with algorithms

```
c.begin() = start

c.end() = 1 past the last element
```

Different kinds of iterator

- random access (i += 3; --i; ++i)
- bidirectional (++i, --i), store/retrieve
- forward (++i), store/retrieve
- input (++i) retrieve
- output (++o) store

Practical iterators

- **iterator** "Standard", goes from beginning to end c.begin(), c.end()
- **const_iterator** Like iterator, but changes can't be made (prefer!) c.begin() and c.end() are overloaded so you can use them to assign their result to a const_iterator
- **reverse_iterator** Goes from the end to the beginning with same semantics as iterator Generally, c.rbegin() and c.rend() list, vector, deque, map, multimap, set, multiset, hash_, string

Conclusion

- The STL has everything
- Let the compiler do the work for you
- Saves time and lines of code
- Run-time efficiency of the code that is generated