# Programmazione generica

Luca Lista

*INFN, Sezione di Napoli*

# Programmazione generica

Il C++ fornisce un metodo per creare un *polimorfismo parametrico*. E' possibile utilizzare lo stesso codice per tipi differenti: il tipo della variabile diventa un parametro.

```
template<class T>
T max( T p1, T p2 ) {
  if ( p1 < p2 ) return p2;

  return p1;
}
```

Per il tipo T deve essere definito l'operatore <

Main.cc

```
int main() {
 Vector v1,v2;
 cout << max(10,20) << endl;
 cout << max(2.6,1.0) << endl;
 cout << max(v1,v2) << endl;}
```

# Templates

- La stessa definizione di funzione si può applicare a tipi diversi

```
template<class T>
T sqr(const T x)
{
  return x * x
}
```

```
int i = 1, j = 3;
int n = max( 3, 2 );

String a = “Mela”,
       b = “Pera”;
String c = max( a, b );

Matrix m;
Matrix mSqr = sqr( m );
```

```
template<class T>
T min(const T a, const T b)
{
  return (a < b ? a : b);
}

template<class T>
T max(const T a, const T b)
{
  return (a > b ? a : b);
}
```

# Sintassi

- `template < class` *`identifier`* `> function definition`
- `template < class` *`identifier`* `> class definition`

Ogni volta che nella definizione della funzione o della classe appare *identifier* questo viene sostituito dal compilatore con il tipo fornito nella chiamata.

La dichiarazione e l'implementazione del template devono essere nello stesso file ove il template viene utilizzato

# Un esempio: lo stack di interi

User code

```
int main() {
Stack s;
s.push ( 10 );
s.push ( 20 );
cout << s.pop() << " - " << s.pop();
return 0;
};
```

```
class Stack {
public:
  Stack() {top = NULL;}
  ~Stack() {;}
  void push ( int i ) {
    Contenuto* tmp = new
      Contenuto(i,top );
    top = tmp; }
  int pop () {
    int ret = top->GetVal();
    Contenuto* tmp = top;
    top = top->GetNext();
    delete tmp;
    return ret;
  }
private:
  Contenuto* top;
};
```

```
class Contenuto {
public:
  Contenuto ( int i, Contenuto* ptn ) {
              val=i; next=ptn; }
  int GetVal (){ return val; }
  Contenuto* GetNext() {return next;}
private:
  Contenuto* next;
  int val;
};
```

Output

```
>> 20 - 10
```

# Lo stack “*templizzato*”

User code

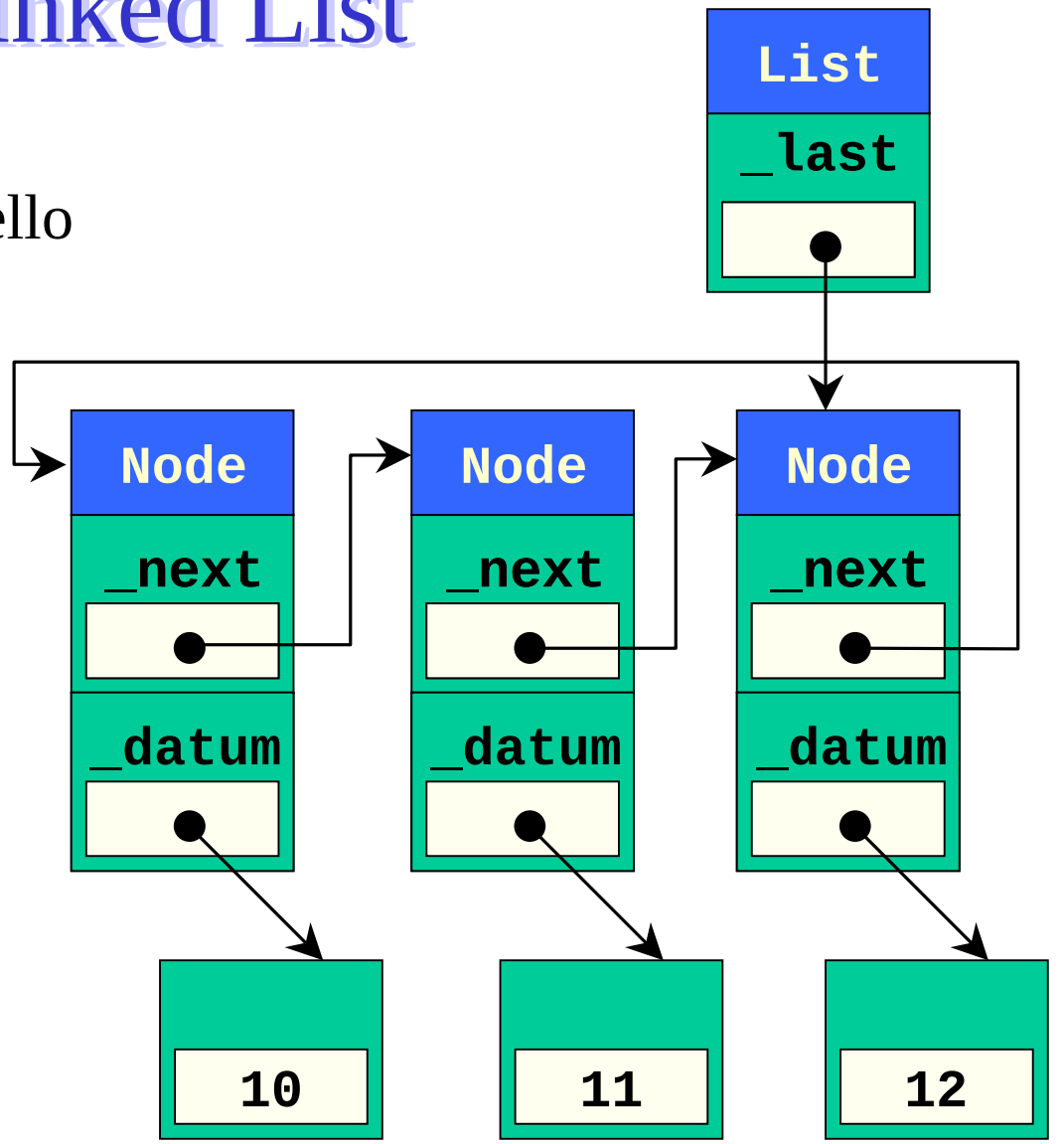
```
int main() {
Stack<int> s;
s.push ( 10 );
s.push ( 20 );
Stack<double> s1;
Stack<Shape*> s2;
cout << s.pop() << “ “ << s.pop();
return 0;};
```

```
template <class T>
class Stack {
public:
  Stack() {top = NULL;}
  ~Stack() {;}
  void push ( T i ) {
    Contenuto<T>* tmp = new
      Contenuto<T> (i,top );
    top = tmp; }
  T pop () {
    T ret = top->GetVal();
    Contenuto<T>* tmp = top;
    top = top->GetNext();
    delete tmp;
    return ret;
  }
private:
  Contenuto<T>* top;
};
```

```
template <class T>
class Contenuto {
public:
  Contenuto ( T i, Contenuto*
ptn ) { val = i; next = ptn; }
  T GetVal (){ return val; }
  Contenuto<T>* GetNext()
{return next;}
private:
  Contenuto* next;
  T val;
};
```

# Linked List

- Contenitore di oggetti dello stesso tipo
- Ogni nodo ha un puntatore al successivo; l'ultimo nodo punta al primo, e chiude la catena



# Linked List: interfaccia

```
#ifndef LIST_H
#define LIST_H

template<class T>
class List
{
public:
  List();
  ~List();
  void append(T *x);
  T* operator[](int) const;
  int length() const;
protected:
  class Node
  {
  public:
    Node(T *x) : _next(0), _datum(x) {}
    Node* _next;
    T* _datum;
  };
  Node* _last;
  int _length;
  friend class ListIterator<T>; // vedi dopo...
};

#endif
```

direttive al preprocessore
per evitare inclusione multipla

# Linked List: implementazione

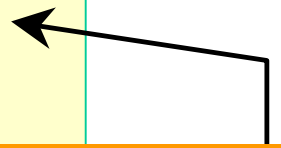
```
template<class T>
List<T>::List() :
  _last(0), _length(0)
{}

template<class T>
void List<T>::append(T *x)
{
  if (_last == 0)
  {
    _last = new Node(x);
    _last->_next = _last;
  }
  else
  {
    Node* first =
      _last->_next;
   _last->_next =
      new Node(x);
   _last = _last->_next;
   _last->_next = first;
  }
  _length++;
}
```

```
template<class T>
int List<T>::length() const
{ return _length; }

template<class T>
T* List<T>::operator[](int i) const
{
  if (i >= _length || i < 0)
    return 0;
  else
  {
    Node* node = _last->_next;
   while(i-- > 0)
     node = node->_next;
   return node->_datum;
  }
}

template<class T>
List<T>::~List()
{
  if (_last == 0) return;
  while (_last != _last->_next)
  {
    Node* last = _last;
    _last = _last->_next;
    delete last;
  }
  delete _last;
}
```



**L'uso di [ ] è inefficiente**

# Linked List: uso

```
#include <iostream>
#include "List.h"

int main()
{
  List<int> list;
  int i = 10, j = 100, k = 1000;

  list.append(&i);
  list.append(&j);
  list.append(&k);

  // uso inefficiente!
  for(i = 0; i < 3; i++)
  { cout << *list[i]; }

  return 0;
}
```

# Iteratore sulla lista

```
#ifndef LISTITERATOR_H
#define LISTITERATOR_H
#include "List.h”

template<class T>
class ListIterator
{
public:
  ListIterator(const List<T>& list);
  T* current() const;
  bool next();
  void rewind();
private:
  const List<T> * _list;
  List<T>::Node* _current;
};

#endif
```

Aggiungere:

```
friend class ListIterator<T>;
```

all’header file `List.h` !

# Iteratore sulla lista

```
template<class T>
ListIterator<T>::ListIterator
  (const List<T>& list) :
  _list(&list), _current(0)
{}

template<class T>
T* ListIterator<T>::current
  () const
{
  if (_current == 0)
    return 0;
  else
    return _current->_datum;
}

template<class T>
void ListIterator<T>::rewind()
{ _current = 0; }
```

```
template<class T>
bool ListIterator<T>::next()
{
  // lista vuota
  if (_list->_last == 0)
    return false;

  if (_current == 0)
  {
    _current =
      _list->_last->_next;
    return true;
  }
  else
  {
    _current =
      _current->_next;
    if(_current !=
       _list->_last->_next)
      return true;
    else
    {
      return false;
    }
  }
}
```

# Iteratore sulla lista

- Uso di **`List`** e **`ListIterator`**

```
#include <iostream>
#include "List.h"
#include "ListIterator.h”

int main()
{
  List<int> list;
  int i = 10, j = 100, k = 1000;
  list.append(&i);
  list.append(&j);
  list.append(&k);

  // uso inefficiente!
  for(i = 0; i < 3; i++)
  { cout << *list[i]; }

  // iterazione efficiente
  ListIterator<int> it(list);
  while(it.next())
  {
    cout << *(it.current()) << endl;
  }

  return 0;
}
```