

La programmazione *Object Oriented* e le sue applicazioni in Fisica.

Gabriella Cataldi, INFN Lecce

Daniele Martello, dip. Fisica & INFN Lecce

make
make

Makefile
Makefile

Un metodo per
automatizzare
il processo di
compilazione



```
emac: Makefile
File Edit Mule Apps Options Buffers Tools Makefile Help

# define the compiler
CXX = g++

# define compilation flags to
CXXFLAGS = -g

# define the loader, and the loading flags
LD = $(CXX)
LDFLAGS =

# find out in which directory we are
LOCALDIR=$(shell basename `pwd`)
THISDIR=$(notdir $(LOCALDIR))

# define sources to be all file with extension .cc
SOURCES := $(wildcard *.c*)

# define objects to be all files with the same name of sources but extension .o
OBJECTS := $(SOURCES:.c*.o)

#
# R U L E S
#

# default is build all the libraries
all: $(OBJECTS)
$(LD) $(CXXFLAGS) $(LDFLAGS) Main.cpp *.o -o $(THISDIR)

# compile a .cc
%.o : %.c
echo I compile source $@
$(CXX) $(CXXFLAGS) -c $< -o $@

IS08-----XEmacs: Makefile (Makefile Font)-----All-----
```

definizione di variabili

stanze

Programmazione Generica: *classi template*

Proviamo a costruirci uno **stack**.

Uno stack e' un elemento del codice all'interno del quale e' possibile salvare oggetti (nell'esempio interi) ed estrarli seguendo la logica del *first-in last-out*. Uno stack si puo' comparare a una pila di piatti.

```
class IntStack {  
    int ssize = 100 ;  
    int stack[ssize];  
    int top;  
public:  
    IntStack() : top(0) {}  
    void push(int i) {  
        if (top > ssize) cout<< "Too many push()es" <<endl;  
        stack[top++] = i;  
    }  
    int pop() {  
        if (top < 0) cout<< "Too many pop()s" <<endl;  
        return stack[--top];  
    }  
};
```

IntStack usa interi! Se mi occorre uno stack di altri tipi di oggetti sono costretto a riscrivere il codice!!!

Programmazione Generica: *classi template*

In C++, esiste un modo per implementare la parametrizzazione del tipo di oggetti utilizzato.

1
3
4
6

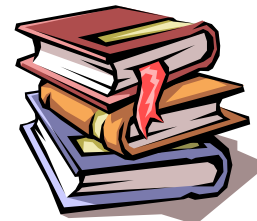
int

1.5
6.8
9.3
5.4

double

12.3 0.1
13.5 0.2
10.4 0.1
11.6 0.3

ErrorPoint



libri

Il *template* permette di *dire* al compilatore che la classe utilizza dei tipi non specificati.

Nel momento in cui il codice di utilizzo della classe viene scritto bisogna specificare il *tipo di oggetto* in modo che il compilatore possa sostituirlo al parametro.

Programmazione Generica: *classi template*

Array.h

```
#include <iostream>
using namespace std;
template<class T> ← Classe template
class Array {
    int size = 100 ;
    T A[size]; ← T parametro di sostituzione
public:
    T& operator[](int index) {
        if(index >= 0 && index < size)
            cout<< "Index out of range" << endl;
        return A[index];
    }
};
```

Main.cpp

```
#include "Array.h"
int main() {
    Array<int> ia;
    Array<float> fa;
    for(int i = 0; i < 20; i++) {
        ia[i] = i * i;
        fa[i] = float(i) * 1.414;
    }
    for(int j = 0; j < 20; j++)
        cout << j << ": " << ia[j]
            << ", " << fa[j] << endl;
}
```

**Specifica
il tipo**

Programmazione Generica: *classi template*

In main si specificano i
tipi di Array

```
int main() {  
    Array<int> ia;  
    Array<float> fa;  
    .....
```

```
template <T>  
class Array_int{  
    int size = 100 ;  
    int A[size];  
public:  
    int& operator[](int index) {  
        .... }  
};
```

```
template <T>  
class Array_float{  
    int size = 100 ;  
    float A[size];  
public:  
    float& operator[](int index) {  
        .... }  
};
```

Il compilatore espande la
classe template Array, per
creare due classi separate
Array_int ed *Array_float*,
sostituendo il tipo al
parametro T

Le Standard Template Library

In molte circostanze si incontra la necessita' di generare nel proprio codice liste di oggetti, senza sapere a priori di quanti oggetti si ha bisogno.

Ricorrere al dimensionamento dinamico non sempre e' una soluzione soddisfacente in quanto il problema di conoscere a priori di quanti oggetti si ha bisogno viene semplicemente demandato a *run time*.

Data.cxx

```
// default constructor. Read the array of data
Data::Data() {
    cout<< "Di quante misure sperimentali hai bisogno? ";
    cin>> _size;
    _data = new ErrorPoint[_size];
    for (int i=0; i<_size; i++) {
        cout<< "inserisci la misura numero " <<i+1<< " nel formato: valore "
            << "(spazio) errore" <<endl;
        cin>> _data[i];
    }
};
```

E se volessi inserire
le misure mentre le
sto prendendo....?



Standard Template Library: Containers

La soluzione al problema e' costituita dalle classi *container* delle Standard Template Library (STD).

Le C++ STD sono state introdotte per risolvere un gran numero di problemi riguardanti la necessita' di gestire insiemi di oggetti e l'implementazione di algoritmi di uso frequente. Esse hanno il vantaggio di essere estremamente flessibili.

Per il momento considereremo solo le classi container

Containers

- *Contenitori generici:*

vector- rapido accesso random agli elementi

deque- rapida allocazione di nuovi elementi ad inizio/fine

list- rapido inserimento di elementi nel mezzo e riordinamento

Sequenze lineari
Sequenze lineari

- *Come ci muoviamo nei containers?*

Iteratori

puntatori “**intelligenti**” che selezionano gli elementi di un container.

Il container attraverso l’iteratore diventa una semplice sequenza che puo’ essere percorsa senza che l’utente debba preoccuparsi della sua struttura.



Un semplice vettore

Main.cxx

```
#include <vector>

int main() {

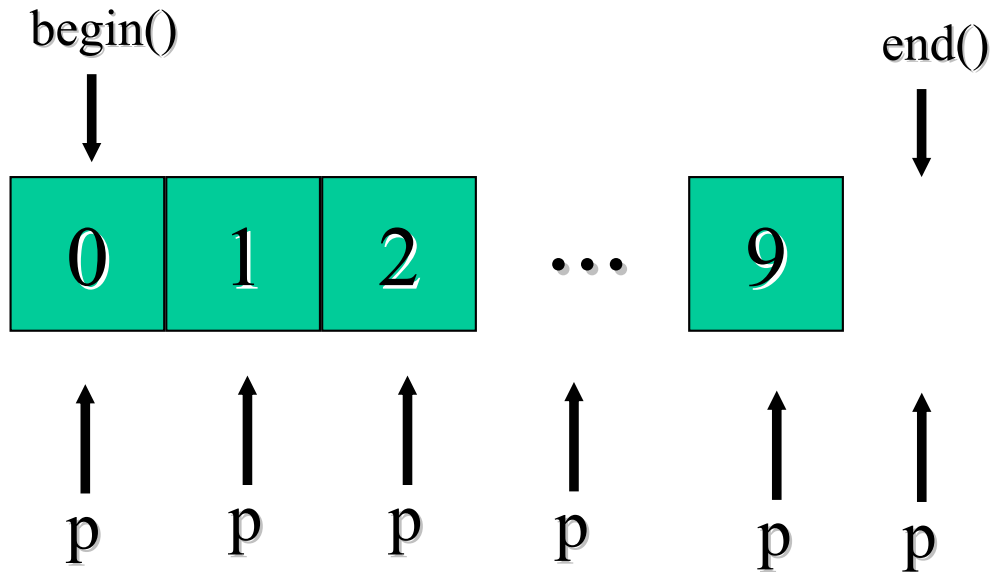
    vector<int> v;                                // crea un vettore di interi vuoto
    cout << v.size() << endl;                    // stampa dimensione di v: zero
    for ( int i = 0; i != 10; ++i )              // un loop da 0 a 9
        v.push_back( i );                       // aggiunge un intero in coda a v

    cout << v.size() << endl;                    // stampa dimensione di v: 10
                                           // crea un iteratore costante per un vettore di interi:
                                           // p si comporta come un "const int *" a tutti gli effetti
    vector<int>::const_iterator p;
    for ( p = v.begin(); p != v.end(); ++p ) // "itera dall'inizio alla fine di v"
        cout << (*p) << " ";
    cout << endl;

    return 0;
}
```

Un semplice vettore

Le funzioni `begin()` e `end()` restituiscono iteratori .



`push_back()` inserisce un nuovo elemento alla fine della sequenza.

Standard Template Library: Esercizio

Data.h

```
#include <deque>
...
typedef deque<ErrorPoint> ErrorArray;
typedef deque<ErrorPoint>::const_iterator const_ErrorIter;
...
class Data {
private:
    ErrorArray _data;
public:
    // return the number of error points in the data sample
    int size() const;
    // return the begin of the data array
    ErrorIter begin();
    // return the end of the data array
    ErrorIter end();
    // return the begin of the data array with a const iterator
    const_ErrorIter begin() const;
    // add a element
    void push_back(const ErrorPoint );
```

Si usa **typedef** quando i tipi di dati diventano complicati o lunghi da scrivere per creare un alias ad un tipo.

Implementazione di Data.h Data.cxx usando un container della STD

Standard Template Library: Esercizio

// constructor the array of data from file

```
Data::Data(const char * filename) {  
    ifstream inputfile(filename);  
    ErrorPoint tmp;  
    while (!inputfile.eof()) {  
        inputfile>>tmp;  
        _data.push_back(tmp);  
    }  
    inputfile.close();  
};
```