Introduzione al C++

prof. Riccardo Torlone
Università di Roma Tre

Il C++

- Il linguaggio C++ nasce nei laboratori di ricerca della AT&T a partire dal 1980 (Bjarne Stroustrup)
- Rappresenta una estensione del linguaggio di programmazione C. Per questo motivo, un compilatore C++ è in grado anche di compilare programmi C
- Le principali estensioni sono:
  - alcuni miglioramenti rispetto al linguaggio C;
  - un supporto per la gestione dei tipi di dato astratto;
  - un supporto per la programmazione orientata agli oggetti.
- Anche per il linguaggio C++ (come il linguaggio C) esistono diverse versioni anche se è in corso di definizione una standardizzazione (ANSI)

Un semplice programma C++

```
// Programma per il calcolo del fattoriale
#include <iostream.h>
int Fatt(int n) { /* funzione che calcola il fattoriale del numero in ingresso */
        int fat;
        fat = 1;
        while (n>=1) {
                fat = fat * n;
                n = n - 1;
        }
        return fat; // valore restituito
}
main() {
        int i;
        cout << “Calcolo del fattoriale di :”;
        cin >> i;
        cout << “Risultato = “; cout << Fatt(i); cout << endl;
}
```

Alcune caratteristiche del linguaggio

- Il programma ha diverse componenti:
  - direttive (iniziano con il simbolo #)
  - definizioni (di variabile o altro)
  - funzioni (tra cui la funzione speciale **main**)
  - classi (le vedremo più avanti)
- La direttiva **#include <iostream.h>** include la libreria delle funzioni predefinite di input/output tipiche del C++
- Le parentesi graffe delimitano un blocco; ciò che è dichiarato in un blocco è visibile solo all'interno del blocco;
- Esistono due modi di specificare i commenti:
  - o sono racchiusi tra /* e */
  - oppure un commento comincia con // e si estende fino al termine della riga;

Altre caratteristiche

- Una variabile viene dichiarata scrivendo il tipo seguito dal nome della variabile
- Una funzione viene specificata scrivendo, nell'ordine, il tipo restituito, il nome della funzione, la lista dei parametri *formali* ed un blocco di istruzioni
- Una funzione viene invocata specificando i parametri *attuali* che devono corrispondere in numero, ordine e tipo ai parametri formali
- Ogni istruzione è conclusa dal simbolo "**;**"
- La funzione speciale **main()** è quella che viene eseguita al lancio del programma
- Le istruzioni di input e di output vengono effettuate tramite gli operatori **>>** e **<<**, rispettivamente.
- I nomi **cin** e **cout** e identificano i dispositivi standard di ingresso/uscita, cioè tastiera e video.

Espressioni e istruzioni in C++

- Una espressione in C++ può essere una costante, una variabile una espressione composta
- Una espressione composta è formata da un operatore e da uno o due operandi che sono a loro volta espressioni.
- Alcuni operandi di C++:
  - Aritmetici: * / % + –
  - Logici: && || ! < <= == != >= >
  - Assegnamento: = *= += –= /=
  - Incremento/decremento: ++ ––
  - If aritmetico: ? :
  - Indirezione: & *
- Una istruzione semplice è una espressione seguita da ";"
- Una istruzione composta è una sequenza di istruzioni racchiuse dai simboli "{" e "}"

Un'altra versione del programma per il calcolo del fattoriale

```
#include <iostream.h>
int Fatt(int n) {
        int fat=1; // inizializzazione all'atto della definizione
        while (n>=1) {
                fat *= n;
                n--;
        }
        return fat;
}
main() {
        int i;
        cout << "Calcolo del fattoriale di :";
        cin >> i;
        cout << "Risultato = " << Fatt(i) << endl;
        /* NB: occorrenze successive degli operatori di input/output si
                possono concatenare */
}
```

Istruzioni di controllo in C++

Sono praticamente le stesse del C e Java. Ricordiamo alcune istruzioni tra le più usate:

- Istruzioni condizionali:
  - **if** (<espressione>) <istruzioni1> **else** <istruzioni2>
  - **switch** (<espressione>) {
    case <costante1> : <istruzioni1>
    case <costante2> : <istruzioni2>
    ....
    default : <istruzioni>
    }
- Cicli:
  - **while** (<espressione>) <istruzioni>
  - **do** <istruzioni> **while** (<espressione>)
  - **for** (<espr1> ; <espr2> ; <espr3>) <istruzioni>
    - (è possibile dichiarare una variabile in <espr1>)

Tipi di dato base in C++

Sono gli stessi del Java e C con alcune varianti

- Tipi primitivi principali: **int short long float double long double char bool**
  - Esempio dichiarazione variabili:

```
int x;
float a, b=1.34;
bool trovato=false;
char carattere='b';
```

- Tipi enumerati: enum
  - Esempio:

```
enum mese {gennaio, febbraio, marzo, ... }
mese m = marzo;
```

Scope di una variabile

- Scope (visibilità) di una variabile: porzione di codice in cui la variabile dichiarata è riferibile (utilizzabile)
- Regola: lo scope di una variabile o di un qualsiasi altro identificatore si estende dal punto immediatamente successivo la dichiarazione fino alla fine del blocco di istruzioni (delimitato dalle parentesi graffe) in cui è inserita

```
// Qui x non e` visibile
... {
  ...       // qui x non e` visibile
  int x = 5;    // Da ora in poi esiste una variabile X
  ...           // qui x e` visibile
  {         // x e` visibile anche in questo blocco
              ...
              }
  ...
} // da questo punto x ora non e` piu` visibile
```

Altre regole di visibiltà

- All'interno di uno stesso blocco non è possibile dichiarare più volte lo stesso identificatore, ma è possibile ridichiararlo in un blocco annidato; in tal caso la nuova dichiarazione nasconde quella più esterna che ritorna visibile non appena si esce dal blocco ove l'identificatore viene ridichiarato

```
{ // blocco A
    int x = 5,y=0; // variabili locali al blocco A (e ai blocchi nidificati in A)
    cout << x << endl;        // stampa 5
    while (y++<3) { // blocco B
        cout << x << ' ';     // stampa 5
        char x = '-';
        cout << x << ' ';     // ora stampa  -
    }
   cout << x << endl;         // stampa di nuovo 5
}
```

- Una variabile **globale** è visibile a livello di file

Conversione del tipo dei dati

Il tipo di un dato può essere convertito implicitamente (con una assegnazione) o esplicitamente (con l'operazione di "cast").

- Esempio:

```
int x=3;
float y = 5.3;
x = int(y); // oppure x = (int)y;
y = x;
```

Costanti in C++

- Si definiscono facendo precedere la parola chiave **const** dalla definizione di una variabile
- La costante deve essere inizializzata

- Esempio:

```
const float PIGRECO = 3.14;
const int N = 100;
```

Alias o riferimenti

- Sono riferimenti ad una una variabile già definita
- Si fa uso nella dichiarazione del carattere speciale **&**

- Esempio:

```
int x;
int& y = x;
x = 3;
y = y++;
cout << x; // stampa 4
```

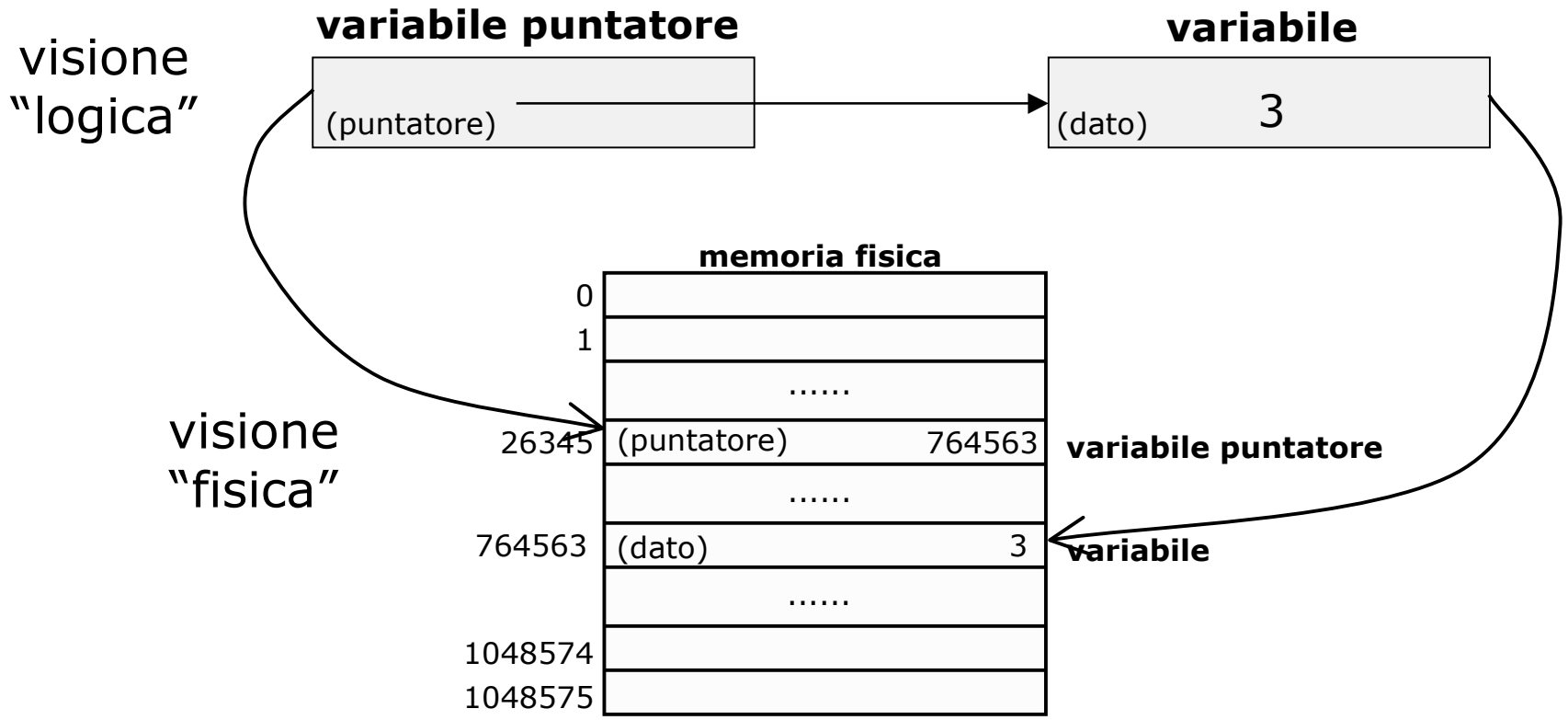
Record in C++

- E' possibile definire dati strutturati con il tipo **struct**
- Per un dato strutturato è possibile definire anche operazioni da effettuare su di esso

```
struct data {
        int giorno;
        int mese;
        int anno;
        void stampa() {
                cout << giorno << "/" << mese << "/" <<  anno
        }
};
...
data oggi;
oggi.giorno = 6; oggi.mese = 3; oggi.anno= 2002;
oggi.stampa();
```

Puntatori

- Un **puntatore** è una variabile che contiene un riferimento ad un'altra variabile
- Da un punto di vista fisico un **puntatore** è l'indirizzo di una locazione di memoria (RAM)



Definizione di un puntatore

- In C un puntatore si dichiara anteponendo al nome di una variabile il simbolo *****

- Esempi di definizione di puntatori:

```
int *i;
char *c;
float *n;
int* j;
float *n1, *n2;
```

Operazioni sui puntatori

- L'operatore **&** applicato ad una variabile restituisce il puntatore ad essa
- L'operatore ***** applicato a un puntatore restituisce la variabile puntata
- Esempi:

```
int i, j;
int *p,*q;
p = &i;   /* p punta alla variabile i */
j = *p;    /* nella variabile j va il contenuto della variabile
              puntata da p */
*q = j;    /* nella variabile puntata da q va il contenuto di j */
*p = *q; /* nella variabile puntata da p va il contenuto della variabile
              puntata da q */
p = q;    /* p e q puntano alla stessa variabile */
```

Allocazione statica di variabili (e oggetti)

- In C++ è possibile creare (allocare) variabili (e oggetti!) in maniera **statica** o **dinamica**
- Nell'allocazione statica una variabile esiste (è utilizzabile) dal momento della sua dichiarazione fino al termine del blocco nel quale è stata dichiarata (scope della variabile)

```
int Fatt(int n) {
  int fat=1; // inizio esistenza variabile fat (allocazione)
  while (n>=1) {
            fat *= n;
            n--;
  }
  return fat;
}// fine esistenza fat (deallocazione a carico del sistema)
```

Allocazione e deallocazione dinamica di variabili

- Nell'allocazione dinamica una variabile viene allocata e deallocata esplicitamente con opportuni operatori (**new** e **delete**)
- Operativamente la gestione avviene mediante l'uso di puntatori e della costante simbolica predefinita NULL

```
int *p1=NULL;
p1 = new int; // allocazione di una variabile intera puntata da p1
*p1=4; // uso della variabile allocata dinamicamente
delete p1; // deallocazione esplicita della variabile puntata da p1
```

- L'allocazione dinamica si usa tipicamente per la gestione di collezioni di dati la cui cardinalità non è nota a priori

Gestione della memoria a runtime

Durante l'esecuzione di un programma C++ (*runtime*) la memoria viene gestita dal sistema in due maniere

- **gestione statica**: viene allocata dal sistema operativo un'area di memoria di dimensione fissa per tutta l'esecuzione del programma (variabili globali)
- **gestione dinamica**: vengono allocate due aree di memoria di dimensioni variabili che vengono usate in maniera diversa:
  - lo stack: quando una funzione viene invocata vengono allocate automaticamente tutte le variabili locali e i parametri attuali sullo stack in un *record di attivazione*; successivamente, quando l'esecuzione della funzione termina, il record di attivazione viene cancellato e lo stack viene riportato nella situazione in cui era prima dell'invocazione;
  - lo heap: la gestione viene lasciata al programmatore mediante creazione e distruzione dinamica di variabili (tramite puntatori)

Array in C++

- Gli array in C++ si possono definire in maniera statica o dinamica.
- Definizione statica:

```
int vettore[10]; // vettore di interi (10 elementi)
int matrice[2][3]; // matrice 2X3 di interi
for (int i=0; i < 10; i++) vettore[i] = 0; /* gli indici vanno da 0
                                              a N-1 */
matrice [1][2] = 3;
float x[3] = {1.0, 2.3, 3.4 } /* Un array può essere inizializzato
            all'atto della sua creazione */
float y[] = {1.0, 3.3, 5.0, 6.3 }     /* In questo caso la sua
  dimensione viene calcolata automaticamente */
int mat[2][2] = { {1,2}, {3,4}  }
matrice[0][1]= matrice[1][1]*y[3];
```

Array e puntatori

- Esiste uno stretto legame tra array e puntatori
- Definendo: float x[4];
  - x denota un puntatore al primo elemento dell'array,
  - incrementando x di un intero k si accede alla posizione kesima dell'array (ovvero all'elemento di indice k+1)
- Quindi:

  *x e x[0] sono la stessa cosa,

  x e &x[0] sono la stessa cosa,

  *(x + 3) e x[3] sono la stessa cosa,

  (x + 2) e &x[2] sono la stessa cosa

Allocazione dinamica degli array in C++

E' possibile allocare e deallocare dinamicamente vettori in C++ usando opportunamente gli operatori **new** e **delete**

- Esempio:

```
int i;
char *s;
cin >> i;
s = new char[i+1]; /* viene allocato un vettore di
        caratteri di dimensione i+1 */
for (int j=0; j < i; j++) cin >> s[j];
s[i] = '\0';
...
delete [] s; /* L'intero vettore s viene deallocato. */
```

Funzioni: passaggio per valore e per riferimento

- I parametri ad una funzione possono essere passati **per valore** o **per riferimento**
- Nel passaggio per valore (modalità di default) il parametro passato non viene modificato dalla funzione (si crea una copia del parametro e la funzione opera su tale copia)
- Nel passaggio per riferimento (si specifica facendo precedere il parametro formale dal simbolo **&**) il parametro passato può essere modificato dalla funzione (la funzione opera sul parametro stesso)

Esempi di passaggi di parametri a una funzione

```
int quadrato(int x) {
        x = x * x;
        return x;
}
void scambia(int &x, int &y) {
        int z = x;
        x = y;
        y = z;
}
...
int x = 2, y = 3;
cout << quadrato(x); // stampa 4
cout << x; // stampa 2
scambia(x,y);
cout << x; // stampa 3
cout << y; // stampa 2
```

Valore restituito come alias

```
int& max(int v[], int n) {
        int m = 0;
        for(int i=0 ; i<n ; i++)
                if (v[i] > v[m]) m = i;
        return v[m];
}
...
int dati[10] = {4,3,2,3,4,5,6,9,6,7};
cout << max(dati,10); // stampa 9
max(dati, 10) = 15; /*sostituisce 9 con 15 nel vettore dati*/
```

Valori di default per una funzione

```
char* LeggiStringa(int n=10) {
        char* str = new char[n+1];
        for (int i=0; i < n; i++) cin >> str[i];
        s[i+1] = '\0';
        return str;
}
...
char *s1, *s2;
s1 = LeggiStringa(50); /* crea e legge una stringa
                          di 50 caratteri */
s2 = LeggiStringa(); /* crea e legge una stringa
                          di 10 caratteri */
```

Ricorsione

- È possibile definire funzioni in C++ in maniera ricorsiva

```
int Fatt(int n) {
  if (n==0) return 1; else return (n*Fatt(n-1));
}
...
main() {
  int fat,n=0;
  cin << n;
  fat = Fatt(a);
}
```

- La ricorsione viene gestita attraverso l'uso opportuno dello stack: ogni chiamata della funzione ricorsiva genera un nuovo record di attivazione (con memorizzazione dei parametri attuali e allocazione di memoria delle variabili locali) che "maschera" la precedente invocazione

Funzioni inline

- Una funzione in C++ si può dichiarare *inline*
- In questo caso ad ogni sua invocazione il codice viene direttamente inserito nel codice oggetto, senza utilizzare il meccanismo standard di trasferimento di controllo (creazione del record di attivazione, inserimento nello stack...).
- Esempio:

```
inline int max(int a, int b) {
        return( a > b ? a : b);
}
```

- E' un meccanismo che si usa quando si devono realizzare funzioni semplici e non si vuole appesantire con esse il tempo di esecuzione del programma

Overloading di funzioni

```
const int N=3;
int max(int a, int b) { return( a > b ? a : b); }
float max(float a, float b) { return( a > b ? a : b); }
int max(int v[]) {
        int m = 0;
        for(int i=0 ; i<N ; i++) if (v[i] > v[m]) m = i;
        return v[m];
}
main() {
        int x=2,y=3;
        cout << max(x,y) << endl;
        float a=2.3, b=3.5;
        cout << max(a,b) << endl;
        int vett[N];
        for(int j=0; j<N; j++) cin >> vett[j];
        cout << max(vett);
}
```

Definizioni e dichiarazioni

- In C++ si distingue tra **definizione** e **dichiarazione** di una funzione
- Una definizione comporta la specifica completa della funzione (sequenza di istruzioni da svolgere)
- Una dichiarazione specifica invece solo le caratteristiche della funzione (numero e tipo dei parametri di ingresso e uscita)
- La dichiarazione di una funzione è detta anche **prototipo** o **header**

```
int Fatt(int);  //dichiarazione della funzione Fatt
int Fatt(int n) { //definizione della funzione Fatt
  int fat = 1;
  while (n>=1) { fat = fat * n; n = n - 1; }
  return fat;
}
```

Organizzazione dei programmi C++

- In C++ un programma è generalmente distribuito su più file
- Ogni file costituisce una componente (modulo) dell'applicazione
- Ogni modulo è decomposto in due file:
  - Le dichiarazioni vengono poste in file detti *header* che hanno estensione ".h"
    - *con tale file si dichiarano i servizi offerti dal modulo*
  - Le definizioni vengono poste in file che hanno estensione ".C" o ".cpp"
    - *tale file costituisce l'effettiva implementazione dei servizi*

alfa.cpp (definizioni)

```
#include <iostream.h>
#include "alfa.h"
.....
 int f(char a)  {//definizione di f
            ......
 }
```

alfa.h (dichiarazioni)

```
#ifndef ALFA_H
#define ALFA_H
.....
int f(char);//dichiarazione di f
.....
#endif
```

Struttura di un file header

```
// File nomefile.h
#ifndef NOMEFILE_H
#define NOMEFILE_H
< sequenza di dichiarazioni di costanti >
< sequenza di dichiarazioni di tipi >
< sequenza di dichiarazioni di variabili >
< sequenza di dichiarazioni di funzioni >
#endif
```

- L'identificatore NOMEFILE_H è solo un nome che associamo alle dichiarazioni contenute nel file

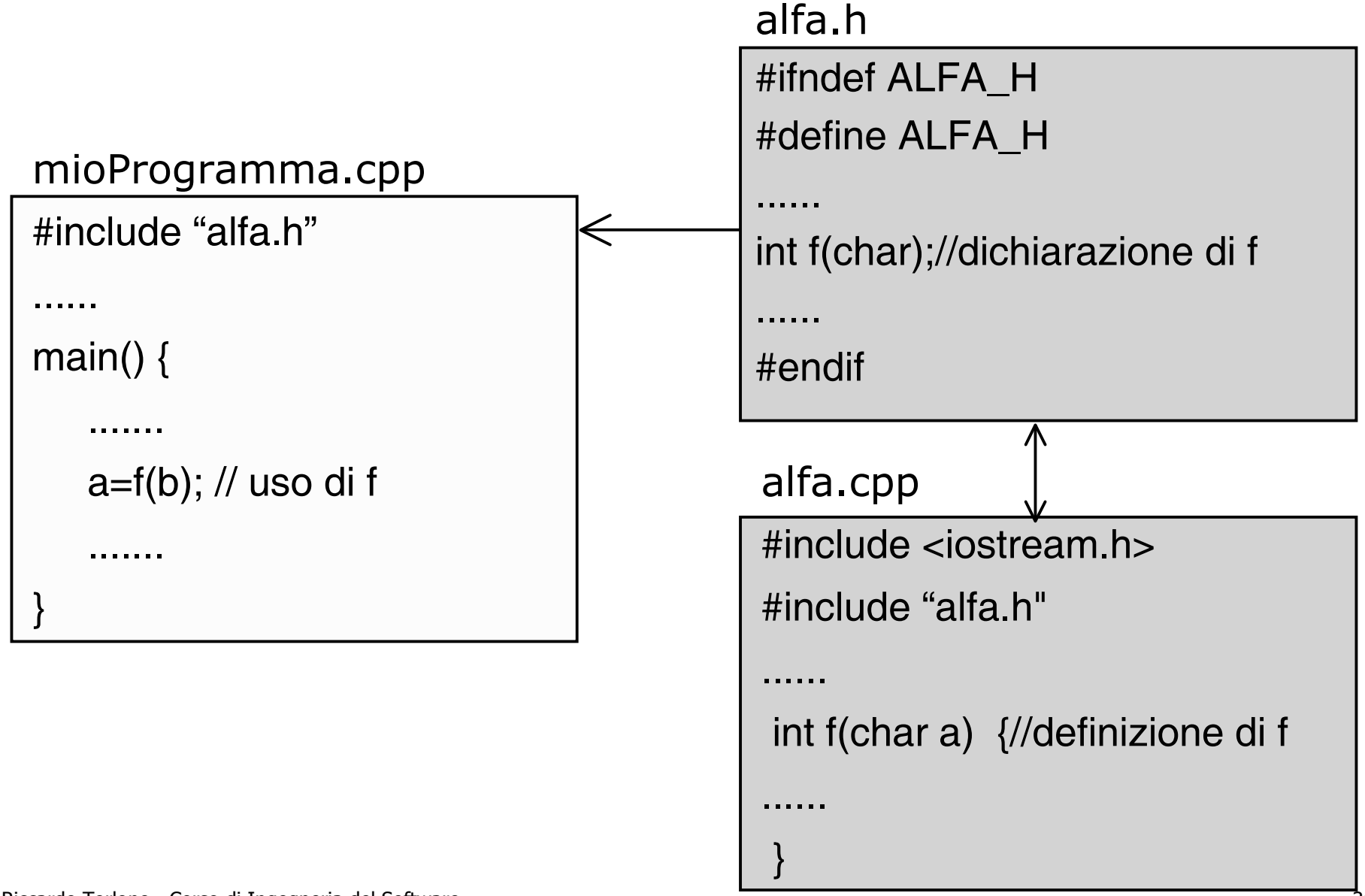
Comunicazione tra componenti separate

- Quando si vuole usare in un file A funzionalità (funzioni, dati e classi) definite altrove, si *includono* in A le loro **dichiarazioni** (contenute nei file header) e si rendono così visibili all'interno di del file A
  - Per includere un file in un altro si usa la direttiva di compilazione:

    #include <..> (librerie di sistema)
  - oppure:

    #include "../" (librerie definite dal programmatore)
- In questa maniera si disacoppia l'uso di una componente dalla sua effettiva implementazione
- Questo consente tra l'altro la compilazione separata delle componenti
- Il **linker** ha poi il compito di associare nella maniera corretta le invocazioni alle definizioni

Organizzazione di programmi su più file



Esempi di semplici programmi C++

```
// Gestione di vettori
#include <iostream.h>
int Somma(int v[], int n) {
        int s = 0;
        for (int i = 0; i < n; i++) s = s + v[i];
        return s;
}
int* CreaLeggiVettore(int n) {
        int* vett = new int[n];
        for (int i = 0; i < n; i++) cin >> vett[i];
        return vett;
}
int& Max(int v[], int n) {
        int m = 0;
        for (int i = 0; i < n; i++) if (v[i] > v[m]) m = i;
        return v[m];
}
```

Gestione di vettori

```
void Scambia(int& i, int& j) {
        int temp = i;
        i = j;
        j = temp;
}

void SelectionSort(int v[], int n) {
        for (int i = 0; i < n - 1; i++) {
                int min = i;
                for (int j = i + 1; j < n; j++)
                        if (v[j] < v[min]) min = j;
                Scambia(v[min], v[i]);
        }
}
```

Main per la gestione di vettori

```
main( ) {
        const int N = 5;
        int *v1 = CreaLeggiVettore(N);
        cout    << "Il massimo e': " << Max(v1,N)  << endl;
        Max(v1,N) = 15;
        cout    << " La somma e': " << Somma(v1,N)  << endl;
        Selection(v1,N);
        for (int i = 0; i < N; i++) cout << v1[i] << ' ';
        cout << endl;
}
```

Gestione di liste collegate di interi

```
#include <iostream.h>
struct RecordLista  {
        int info;
        RecordLista *next;
};
typedef RecordLista * Lista;
Lista LeggiLista()  { // nota: il valore 0 termina l'inserimento
        Lista p = NULL;
        int v;
        cin >> v;
        if (v != 0) {
                p = new RecordLista; // allocazione di un nuovo record
                p->info = v;
                p->next = LeggiLista();
        }
        return p;
}
```

```
boolean Cerca(Lista p, int e) {
        while (p != NULL)
                if (p->info == e) return VERO;
                else p = p->next;
        return FALSO;
}

void StampaLista(Lista p) {
        if (p != NULL) {
                cout << p->info << ' ';
                StampaLista(p->next);
        }
        else cout << endl;
}
```

```
void InvertiLista(Lista& p) {
        Lista temp = NULL, temp1=NULL;
        // temp punta alla testa della lista parzialmente invertita;
        // p punta alla testa della lista ancora da invertire
        while (p != NULL) {
                temp1 = p->next; p->next = temp;
                temp = p; p = temp1;
        }
        p = temp;
}
void main() {
        Lista p = NULL;
        p = LeggiLista();
        InvertiLista(p);
        if (Cerca(p,2)) cout << "2 c'è" << endl;
        StampaLista(p);
}
```

Gestione di matrici

```
#include <iostream.h>
const int N = 10;
typedef int matrice[N][N];
void LeggiMatrice(matrice m, int n) {
        cout    << "scrivi una matrice di ordine "  << n << endl;
        for (int i = 0; i < n; i++)
                for (int j = 0; j < n; j++)
                        cin >> m[i][j];
}
void MoltiplicaMatrici(matrice m1, matrice m2, matrice r, int n) {
        for (int i = 0; i < n; i++)
                for (int j = 0; j < n; j++) {
                        r[i][j] = 0;
                        for (int k = 0; k < n; k++) r[i][j] += m1[i][k] * m2[k][j];
                }
}
```

```
void ScriviMatrice(matrice m, int n) {
        for (int i = 0; i < n; i++) {
                for (int j = 0; j < n; j++) cout << m[i] [j] << ' ';
                cout << endl;
        }
}
main() {
        int n = 3; matrice a,b,c;
        LeggiMatrice(a,n);
        ScriviMatrice(a,n);
        LeggiMatrice(b,n);
        ScriviMatrice(b,n);
        MoltiplicaMatrici(a,b,c,n);
        ScriviMatrice(c,n);
}
```