Inf Gen II

INFORMATICA GENERALE II

Ingegneria delle Telecomunicazioni
Università di Trento

Marco Roveri

roveri@irst.itc.it

Introduzione al C++

(Materiale preso ed adattato da materiale del Prof. M. Benedetti)

AA 2005/2006

MR

C++

- C
  - Evoluzione da due linguaggi pre-esistenti: BCPL e B effettuata da Ritchie.
  - Utilizzato per sviluppare UNIX
  - Utilizzato per sviluppare sistemi operativi moderni (Linux, Windows, Mac OS X, …).
  - Hardware independent
  - Standard creato nel 1989 e modificato nel 1999.
- C++
  - Sovrainsieme del C sviluppato da Stroustrup ai Bell Labs.
  - Fornisce Object Oriented capabilities.
  - Ora linguaggio dominante in accademia e industria (anche se Java sta prendendo piede).

C++ vs Java vs JavaScript

- C++ e Java
  - Linguaggi *compilati*.
  - Linguaggi di programmazione ad oggetti: *completi* e *complessi*.
  - Adatto a progetti di *grandi dimensioni*.
  - *Fully extensible*: programmatori possono creare i propri oggetti e strutture dati.
  - *Strong typing*: i tipi delle variabili devono essere dichiarati e non possono cambiare.
  - Java è "*platform independent*", C++ è *platform dependent*.
- JavaScript
  - Linguaggio *interpretato* "facile da usare".
  - *Limitata* programmazione ad oggetti.
  - *Loose typing*.
  - Adatto a progetti di *piccole* dimensioni.
  - Forte integrazione con pagine HTML.
  - JavaScript come Java è "*platform independent*".

Perchè C++

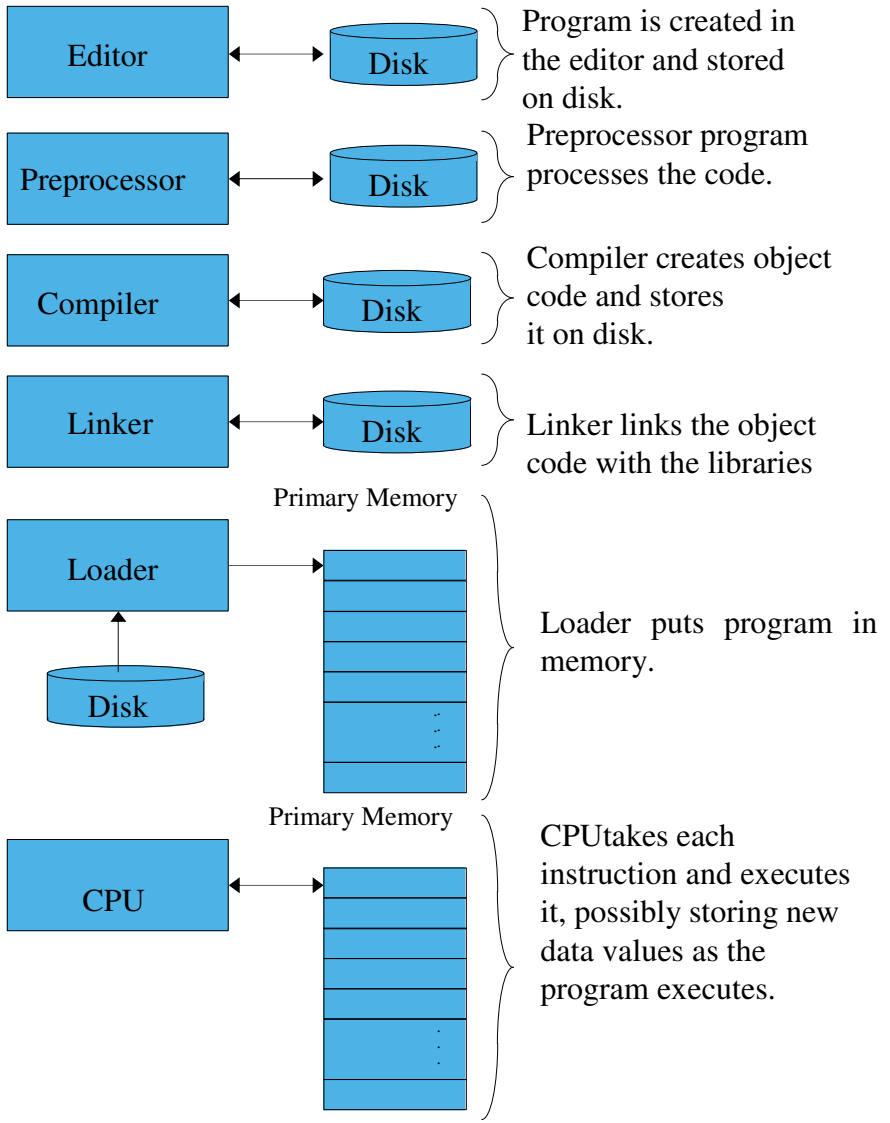
- È fondamentale conoscere più di un linguaggio di programmazione.
- È largamente utilizzato nell'industria e in accademia per la realizzazione di progetti complessi.
- È un linguaggio multi-paradigma (imperativo, object-oriented).
- C++ è un superset del C, più ricco e di più alto livello.
- Il C++ standard comprende una ampia libreria (una volta chiamata Standard Template Library) che fornisce al programmatore una ampia gamma di oggetti generici predefiniti (alcune di queste le vedremo nell'ambito di questo corso).

Programma C++

- Programma C++ è una sequenza *finita* di istruzioni che:
  - Risolvono un problema.
  - Soddisfano i seguenti criteri:
    - Ricevono valori in ingresso.
    - Producono valori in uscita.
    - Sono chiare, non ambigue ed eseguibili.
    - Terminano dopo un numero finito di passi.
  - Operano su strutture dati.

C++:

- Fasi dello sviluppo di un programma C++:
  - Edit
  - Preprocess
  - Compilation
  - Link
  - Load
  - Execute



C++: Lessico

- C++ è *case sensitive.*
- *Blanks*, *Tabs*, *Newline* sono ignorati tranne quando elementi di una stringa.
- *Blanks* possono essere aggiunti per aumentare la leggibilità del codice.
- I commenti del codice come in JavaScript:
  - // solo per i commenti in linea.
  - /* … */ per commenti che possono finire su più linee.
- Keywords, sono parole riservate del linguaggio di programmazione.
- Identificativi sono: nomi di variabili, funzioni, procedure, oggetti.
  - Il primo carattere deve una lettera, un underscore.
  - I caratteri successivi possono essere una lettera, un numero o un underscore.
  - Un identificatore *deve* essere diverso da una qualunque delle keyword.
  - Identificatore: [A-Za-z_][A-Za-z0-9_]*

C++: keywords

break bool case catch char class const continue default delete do double else enum export extern false float for friend goto if inline int long main namespace new NULL operator private protect public return short signed sizeof static std struct switch template this throw true try typeof union using virtual void volatile while

C++: costanti

- Sequenze di caratteri che denotano un particolare valore che non cambia durante l'esecuzione.
- Esempi:
  - Boolean: *true* e *false*
  - Numerica: 5, 2.4, 0xFF, 3E10, -4.5E-9
  - Stringhe: *"marco"*, *"\tstudenti telecomunicazioni\n"*.
  - Primitive: *NULL*

C++: operatori

- Alcuni caratteri speciali e le loro combinazioni sono usati come *operatori*.

```
+    -    *    /    %    ^    &    |    ~
!    =    <    >    +=   -=   *=   /=   %=
^=   &=   |=   <<   >>   >>=  <<=  ==   !=
<=   >=   &&   ||   ++   --   ,    ->*  ->
.*   ::   ()   []   ?:
```

C++: variabili e costanti

- Le *variabili* e le *costanti* sono utilizzate per memorizzare un *valore* in una area di memoria.
- Le variabili consentono la *modifica* del loro valore durante l'esecuzione del programma.
- L'area di memoria corrispondente è identificata da un *nome*, che ne individua l'*indirizzo di memoria.*
- Le variabili e le costanti sono quindi caratterizzate da: *nome* (l'identificatore), *tipo*, *locazione di memoria* (*left-value*) e *valore* (*right-value*).

left-value	right-value
nome	4E10

C++: definizione vs dichiarazione

- **Definizione**: quando il compilatore incontra una definizione di una variabile, esso predispone l'allocazione di un'area di memoria in grado di contenere la variabile del tipo scelto.
  - Sintassi: *tipo identificatore [= expressione];*
  - *Esempio:*
    - ***int*** *x;*
    - ***int*** *x = 3 * 2*
- **Dichiarazione**: specifica solo il tipo della variabile e presuppone dunque che la variabile venga definita in una altra parte del programma.
  - Sintassi: ***extern*** *tipo identificatore;*
  - *Esempio:*
    - ***extern int*** *x;*

C++: dichiarazione di costanti

- Sintassi:

  const *tipo identificatore = expressione;*

  Deve essere possibile calcolare il valore dell'espressione staticamente in fase di compilazione.

- Esempi:

  const int kilo = 1024;

  const double pi = 3.141519;

  const int mille = kilo – 24;

C++: Tipi fondamentali e derivati

- In C++ si distinguono:
  - **Tipi fondamentali**: servono a rappresentare informazioni semplici, come i numeri interi, i caratteri (**bool, int**, **short, long, char, enum, float, double**)
  - **Tipi derivati**: permettono di costruire strutture dati complesse a partire dai tipi fondamentali/derivati (puntatori, riferimenti, array, strutture, union, classi)

C++: operatori

- Manipolano e comparano variabili, costanti, valori.
  - Aritmetici: + - * / % ++ --
  - Assegnazione: = += -= *= /= %=
  - Confronto: == != > >= < <=
  - Logico: ! && ||
  - Bit: & | ^ ~ - << >>
  - Condizione: ?

C++: operatore di assegnazione

- La sintassi dell'operatore di *assegnazione semplice*
  exp1 = exp2
- ***exp1*** deve essere un'espressione dotata di l-value
- ***exp1*** e ***exp2*** devono essere di tipo compatibile
- Il valore denotato da un'espressione di assegnazione è il valore di exp2.
- Un'assegnazione può essere usata all'interno di un'altra espressione.
- L'operazione di assegnazione, '**=**', associa a destra.
- Esempio:
  - **int** a, b, c, d;

    a = b = c = d = 5;
  - equivalente a:

    (a = (b = (c = (d = 5))));

C++: operatori di assegnazione

Forma compatta	Forma Estesa
x += y	x = x + y
x -= y	x = x - y
x *= y	x = x * y
x /= y	x = x / y
x %= y	x = x % y

C++: operatori di incremento

x++	*incrementa* **x** e denota il valore di **x** *prima* dell'incremento
x--	*decrementa* **x** e denota il valore di **x** *prima* del decremento
++x	*incrementa* **x** e denota il valore di **x** *dopo* l'incremento
--x	*decrementa* **x** e denota il valore di **x** *dopo* il decremento

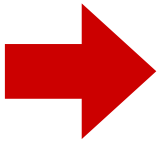
C++: espressioni e istruzioni

- Le **espressioni** sono costrutti che descrivono come calcolare un particolare valore
  - 3 + 2
  - (4 * 3) + 5 == 6
- Gli **statement** o **istruzioni** sono azioni che il processore può elaborare.
  - In C++ uno statement è composto da una o più espressioni.
  - Gli statement sono delimitati dal carattere "**;**"
  - Esempio:
    - y = x + 1;
    - z = (!x || y) && (x || !y);

C++: espressioni

- Valgono le stesse considerazioni viste per JavaScript.
- Le espressioni vengono valutate left to right seguendo l'albero sintattico indotto dalla priorità degli operatori e dalle parentesi.
- Il valore di una espressione può essere calcolato senza valutare l'intero albero sintattico (**lazy-evaluation**)

```
a = 0, b = 0;
(a < b) && (((a + b) + 1) == 0) ?  a : b;
```



false

C++: statement condizionali

- Statement condizionale **if-then**

```
if ( <condizione> ) {
    // Blocco istruzioni se la <condizione> è vera
}
```

- Statement condizionale **if-then-else**

```
if ( <condizione> ) {
    // Blocco istruzioni se la <condizione> è vera
}
else {
    // Blocco istruzioni se la <condizione> è false
}
```

C++: statement condizionali

- Statement condizionale **if-then-else** innestati

```
// Attenzione alle parentesi: i due frammenti di codice sotto sono diversi
if ( <condizione1> ) {
  // blocco istruzioni se <condizione1>  è vera
  if ( <condizione2> ) {
        // Blocco istruzioni se la <condizione2> è vera
  }
  else {
      // Blocco istruzioni se la <condizione2> è false
  }
}

if ( <condizione1> ) {
  // blocco istruzioni se <condizione1>  è vera
  if ( <condizione2> ) {
      // Blocco istruzioni se la <condizione2> è vera
  }
}
else {
    // Blocco istruzioni se la <condizione1> è false
}
```

C++: statement condizionali

- **switch** statement

```
switch ( <integer_expression> ) {
   case v1 : { /* blocco istruzioni */ } break;
   case v2 : { /* blocco istruzioni */ } break;
   ....
   default: { /* blocco istruzioni */ } break;
}
```

C++: ripetizione controllata

- **while** statement

```
while ( <cond> ) {
  /* blocco istruzioni da ripetere */
}
```

- **for** statement

```
for ( <init stm>; <cond>; <inc stm> ) {
  /* blocco istruzioni da ripetere */
}
```

- **do-while** statement

```
do {
  /* blocco istruzioni da ripetere */
} while ( <cond> );
```

C++: array

- Motivazione
  - Supponiamo di dover memorizzare i coefficienti di un polinomio di grado *n*:

$$P(x) = \sum_{i=0}^{n} a_i x^i = a_0 + a_1 x + a_2 x^2 + \cdots + a_n x^n$$

  - Dobbiamo dichiararne tante quanti sono i coefficienti – ognuna con un nome diverso – e poi usarle opportunamente.

    int a0, a1, …., an;

  - Se da un polinomio di grado *n* passiamo a manipolare un polinomio di grado *2n* dobbiamo aggiungere *n* nuove variabili distinte.
  - Inoltre se il grado del polinomio non è noto a priori in fase di compilazione del programma, che cosa posso fare?
  - In matematica, per affrontare situazioni simili, si sceglie un unico nome per il parametro/coefficiente e poi si ricorre alla notazione *con pedice*: si usa la scrittura "a_i" per indicare l'i-esimo coefficiente di un insieme di coefficienti {a_i, *i=0,…,n*}.

C++: array

- In C++ esiste una possibilità del tutto analoga (come in JavaScript):
- Invece di scrivere a_i (il pedice non si può indicare in un file sorgente, essendo un documento di puro testo) in C++ si scrive: **a[i]**
- L'oggetto **a** nell'insieme prende il nome di **array**; un array è quindi un raccoglitore di tante variabili, che hanno un nome formato da una parte iniziale uguale al nome dell'array, e si distinguono poi per il loro *indice* indicato tra parentesi quadre.
- Come ogni altro oggetto C++, prima di poterlo usare un array lo si deve *dichiarare*; ad esempio:

  int a[100]

  dichiara che utilizzeremo un array di nome "a", che contiene 100 variabili di tipo **int;**
- Quando si dichiara un array come "**int a[100]**", il compilatore riserva in memoria uno spazio di memoria sufficiente a memorizzare (in maniera adiacente) 100 variabili di tipo intero.
- Come in JavaScript, gli indici con cui si accederà a tali variabili vanno da **0** a **99**.
- In generale se dichiaro un array di **n** elementi esso corrisponderà a variabili il cui indice è compreso tra **0** e **n-1**.
  - Nota bene: l'indice di un array **NON** è compreso tra **1** e **n**.

C++: array

- I singoli elementi di un array di interi sono - a tutti gli effetti - variabili di tipo intero. Lo stesso vale ovviamente per array di qualunque altro tipo (di boolean, float, char,…). Si può dunque:
  - Assegnare un valore ad una *cella* dell'array:

    ```
    a[3] = 10;
    ```

  - Coinvolgere elementi dell'array in espressioni:

    ```
    a[6] = (a[8] = a[3]*a[3]+2)+1;
    ```

  - Utilizzare variabili (intere) o espressioni per *indicizzare* l'array:

    ```
    int i=1, b=5;
    a[i--] = ++b;
    a[i] = 5*a[i+3]+b;
    a[++b] = ++a[a[i+1]];
    ```

  - Non si può invece operare con assegnazioni tra array;
    - se a e b sono due array di interi, scrivere "a=b;" è un errore.

C++: array statici e dinamici

- Quando si scrive, ad esempio:

  float coef[20];

  è già noto a tempo di compilazione lo spazio necessario a memorizzare l'array: si tratta di 20 volte quello necessario a memorizzare un singolo **float**;
- Il compilatore può dunque senz'altro predisporre lo spazio necessario alla memorizzazione dell'array così come farebbe con ogni altra variabile dichiarata;
- Tuttavia, il C++ - a differenza di altri linguaggi - consente anche la dichiarazione *dinamica* degli array senza variazioni nella sintassi d'utilizzo; nelle dichiarazioni dinamiche la dimensione dell'array è un valore calcolabile solo a tempo d'esecuzione; ad esempio, si può scrivere:

  char c[len];

  per dichiarare l'utilizzo di un array c di caratteri di capacità "len", dove len è una variabile o più in generale un'espressione.
- Questo aggiunge notevole flessibilità al linguaggio; la gestione delle complicazioni aggiuntive (l'array deve trovare spazio in memoria a tempo d'esecuzione e non di compilazione, spazio di una dimensione anch'essa nota solo *durante* l'esecuzione) viene delegata al C++: non se ne occupa il programmatore.
- Vedremo più avanti maggiori dettagli sugli array dinamici.

C++: array - inizializzazione

- Come nel caso della dichiarazione di una variabile anche per gli array si può specificare un valore iniziale per gli elementi dell'array.

  int a[10] = {5, 2, -5, 10, 234, 0, 1, 100, -1, 3};

C++: array di caratteri e stringhe

- Un array di caratteri può essere usato per rappresentare stringhe, e in questo caso esiste il vincolo che l'**ultima posizione dell'array non può rappresentare un carattere significativo, ma deve contenere il valore speciale zero**;
- Tale valore serve a varie funzioni che manipolano stringhe a capire *dove finisce la stringa*. Dunque, il numero di caratteri necessari per memorizzare una stringa lunga **n** caratteri è **n+1**;
- Esiste una forma semplificata di inizializzazione utilizzabile per usare un array di caratteri come stringa. La sua sintassi è (esempio):

  char mia_stringa[] = "Ciao a tutti!";
- Vengono riservati in memoria 14 spazi, e la situazione risultante è:

C	i	a	o		a		t	u	t	t	i	!	
67	105	97	111	32	97	32	116	117	116	116	105	33	0

- Si può stampare questa stringa sullo schermo con il semplice comando:

  cout << mia_stringa;

C++: array di caratteri e stringhe

- Esistono molte funzioni di libreria per manipolare array di caratteri che rappresentano stringhe: funzioni per calcolare la lunghezza di una stringa, per concatenare stringhe, per verificare la presenza di una certa sottostringa all'interno di un'altra stringa, e molte altre ancora. Le vedremo più avanti;
- Questo modo di gestire le stringhe è ereditato dal C, ed ha pregi e difetti. Uno dei difetti principali consiste nella "difficoltà" di gestire dinamicamente la lunghezza delle stringhe. Ad esempio, per concatenare due stringhe ad ottenerne una terza è necessario gestire esplicitamente il calcolo e l'allocazione del nuovo array necessario a contenere il risultato;
- Per ovviare a simili problemi il C++ mette a disposizione un meccanismo più flessibile, la classe "String" (che **non** studieremo in questo corso), che permette di maneggiare le stringhe in modo molto naturale e del tutto simile al modo utilizzato in JavaScript.

C++: array multidimensionali

- Come in JavaScript, in C++ si possono dichiarare array bidimensionali (multidimensionali):

  int a[10][5];

  Corrisponde alla dichiarazione di un array di due dimensioni – matrice – con 10 righe e 5 colonne.
- Gli indici con cui si accede alla matrice vanno, come al solito, da 0 alla dimensione meno uno (da zero a 9 per le righe, da zero a 4 per le colonne, nel caso dell'esempio precedente); per accedere all'elemento in posizione di riga i e di colonna j si usa la sintassi: a[i][j];
- Il numero di variabili intere allocate dalla dichiarazione di un array del tipo

  int a[N][M];

  è pari ad NxM, e il valore iniziale di ogni elemento è, al solito, indefinito;
- E' possibile dichiarare ed usare array a tre o più dimensioni, utilizzando una coppia aggiuntiva di parentesi quadre per ogni dimensione; ad esempio:

  float a[10][5][20];

C++: array multidimensionali

- Gli array bidimensionali possono essere inizializzati in fase di dichiarazione con una scrittura di questo tipo (esempio):

```
int a[4][5] = { { 2, 5, -8, 7, 6  },
                { 3,10,  7, 6, 1 },
                {-1, 8, -8, 5, 3  },
                { 2, 5,  8, 4, 2  } };
```

- Anche se noi accediamo e manipoliamo gli elementi di un array a due dimensioni come se esso fosse a tutti gli effetti una rappresentazione bidimensionale, è chiaro che la sua rappresentazione in memoria è comunque *lineare*; in particolare, il C++ memorizza in memoria le matrici una riga dopo l'altra, in modo che l'array precedente si trova memorizzato come:

a_{00}	a_{01}	a_{02}	a_{03}	a_{04}	a_{10}	a_{11}	a_{12}	a_{13}	a_{14}	a_{20}	a_{21}	a_{22}	a_{23}	a_{24}	a_{30}	a_{31}	a_{32}	a_{33}	a_{34}
2	5	-8	7	6	3	10	7	6	1	-1	8	-8	5	3	2	5	8	4	2

- Vedremo più avanti ulteriori considerazioni sugli array multidimensionali e le relazioni con i puntatori.

C++: funzioni

- Motivazioni
  - Quando si scrivono programmi di una certa complessità, l'organizzazione "piatta" del codice è insoddisfacente;
  - Non solo per motivi di leggibilità e chiarezza sarebbe conveniente poter strutturare il codice sorgente in qualche maniera, ma esistono anche motivazioni di *gestione della complessità*, *modularità del codice*, *riusabilità*, *stringatezza* e *non ridondanza* che ci spingono a farlo;
  - Ad esempio, cosa accadrebbe se dovessi scrivere un programma che in tre punti distinti ha bisogno di calcolare il "numero di primi inferiori ad n"? E' forse necessario riscrivere tre volte in questi tre punti il relativo codice?
  - E cosa accade se mi accorgo di uno sbaglio in una precisa funzionalità del programma? Devo forse andare a ricercare in tutto il sorgente dove e in quanti modi e posti è stata implementata quella funzionalità?
  - La risposta a questi problemi è la *strutturazione del codice sorgente in unità o moduli relativamente isolati e indipendenti chiamati* ***funzioni***.

C++: funzioni

- Dal punto di vista del programmatore, una funzione è un **segmento di codice *sintatticamente ben delimitato*, adibito ad uno specifico compito;**
- Questo segmento di codice - che d'ora in poi chiameremo semplicemente **funzione** - *incapsula* determinate funzionalità, *organizzando, racchiudendo e isolando* in uno specifico punto del programma sorgente tutto il codice relativo a quella funzionalità;
- La funzione è sintatticamente ben delimitata nel senso che sia il compilatore che il programmatore hanno modo di capire facilmente dove inizia e dove finisce il codice relativo ad una certa funzione; inoltre, ogni funzione ha un **nome** univoco che la identifica e la distingue dalle altre;
- Il compito di una funzione è - in generale - quello di calcolare un risultato o **valore di ritorno**, a fronte di specifici **valori di ingresso**; per ogni funzione, devono essere fissati a priori il *tipo* del valore di ritorno e il *tipo* e la *quantità* dei valori forniti in ingresso (detti anche **parametri**);
- Esempio:
  - Definire una funzione di nome "primo" che prende in ingresso *un* valore di tipo *intero* e ritorna *un valore booleano*: il suo compito sarebbe quello di determinare se l'intero in ingresso è primo o meno e restituire *true* nel primo caso, *false* nel secondo.

C++: funzioni

Tre sono i “momenti” fondamentali nella vita di una funzione:

1. **Dichiarazione**: consiste nella definizione della ***segnatura*** (o ***prototipo***), consiste nello specificare:
   a) Il nome della funzione;
   b) Il numero e il tipo degli argomenti;
   c) Il tipo di dato ritornato dalla funzione.
2. **Definizione**: definisce come una funzione *precedentemente dichiarata* calcola il valore di ritorno a fronte dell’input. Ovvero definisce il ***corpo*** (o ***body***) della funzione il cui scopo è quello di manipolare i parametri di input (***parametri formali***).
3. **Invocazione**: avendo a disposizione una funzione già dichiarata, si passano alla funzione degli specifici valori di ingresso (***parametri attuali***), coerenti in tipo e numero con la sua dichiarazione, e si accetta il valore di ritorno calcolato dalla funzione.

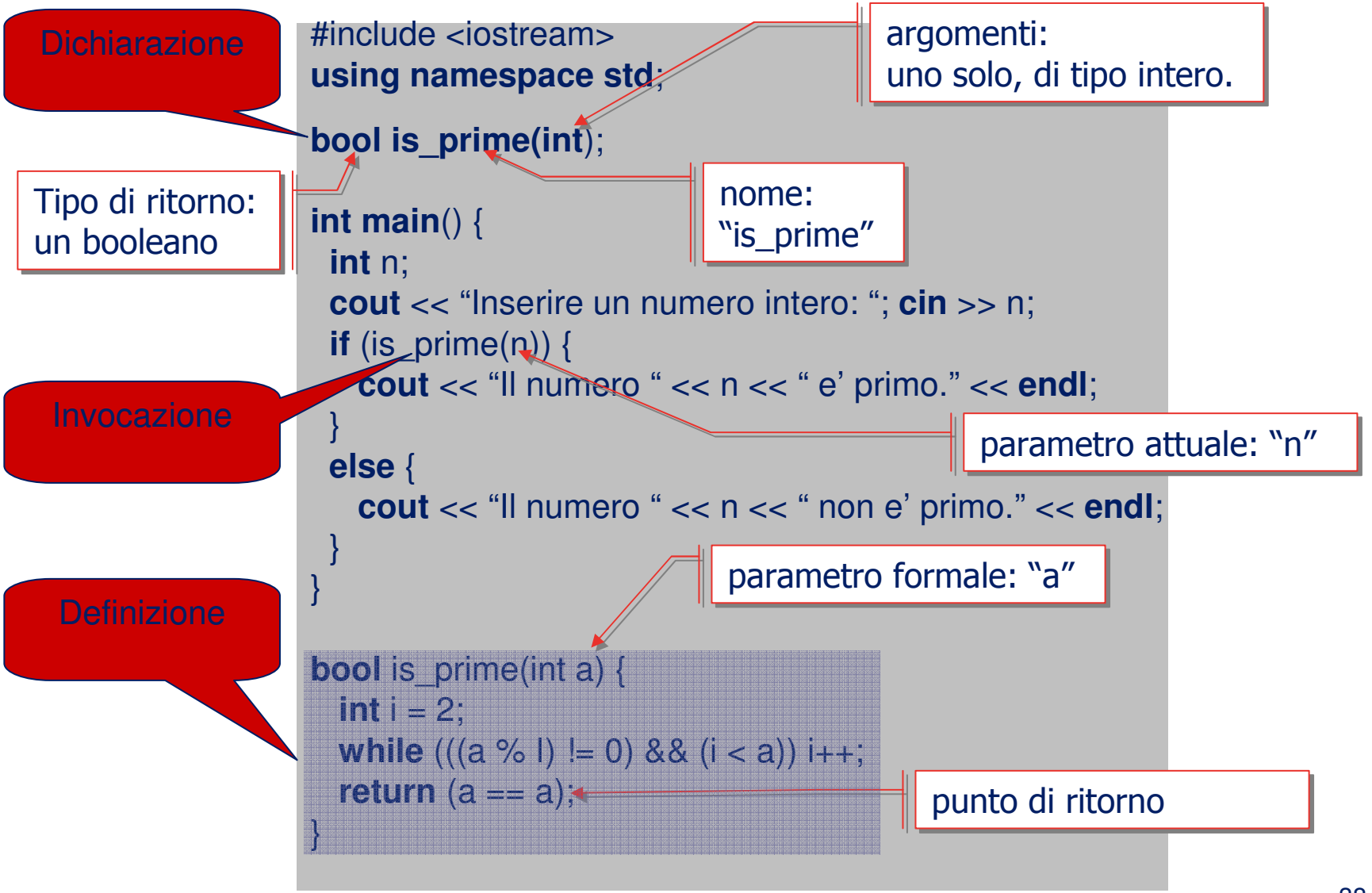
C++: funzioni

- Quando si incontra un'*invocazione* di funzione nel codice - e questo può avvenire in ogni punto di un'espressione in cui è ammesso un valore del tipo di ritorno della funzione - il valore dei *parametri attuali* al momento dell'invocazione viene calcolato e "copiato" - in modo che specificheremo meglio nel seguito - nei *parametri formali* della funzione;
- A questo punto, l'esecuzione del codice in cui si trova l'invocazione viene sospeso, e viene eseguito il codice relativo alla funzione (con i valori dei parametri formali opportunamente inizializzati);
- Quando la funzione ha completato i suoi calcoli e ha pronto il valore da *ritornare al chiamante*, essa segnala la terminazione tramite l'istruzione:

  return exp;

  …dove "**return**" è una parola chiave del C++, mentre **exp** è il valore da ritornare: una qualunque variabile o espressione coerente con il tipo di ritorno della funzione;
- A questo punto, l'esecuzione del codice in cui era stata incontrata l'invocazione della funzione riprende, e "al posto" della funzione c'è ora il suo valore di ritorno.

C++: funzioni


Dichiarazione
Tipo di ritorno: un booleano
Invocazione
Definizione
#include <iostream>
using namespace std;
bool is_prime(int);
int main() {
int n;
cout << "Inserire un numero intero: "; cin >> n;
if (is_prime(n)) {
cout << "Il numero " << n << " e' primo." << endl;
}
else {
cout << "Il numero " << n << " non e' primo." << endl;
}
}
bool is_prime(int a) {
int i = 2;
while (((a % I) != 0) && (i < a)) i++;
return (a == a);
}
argomenti: uno solo, di tipo intero.
nome: "is_prime"
parametro attuale: "n"
parametro formale: "a"
punto di ritorno

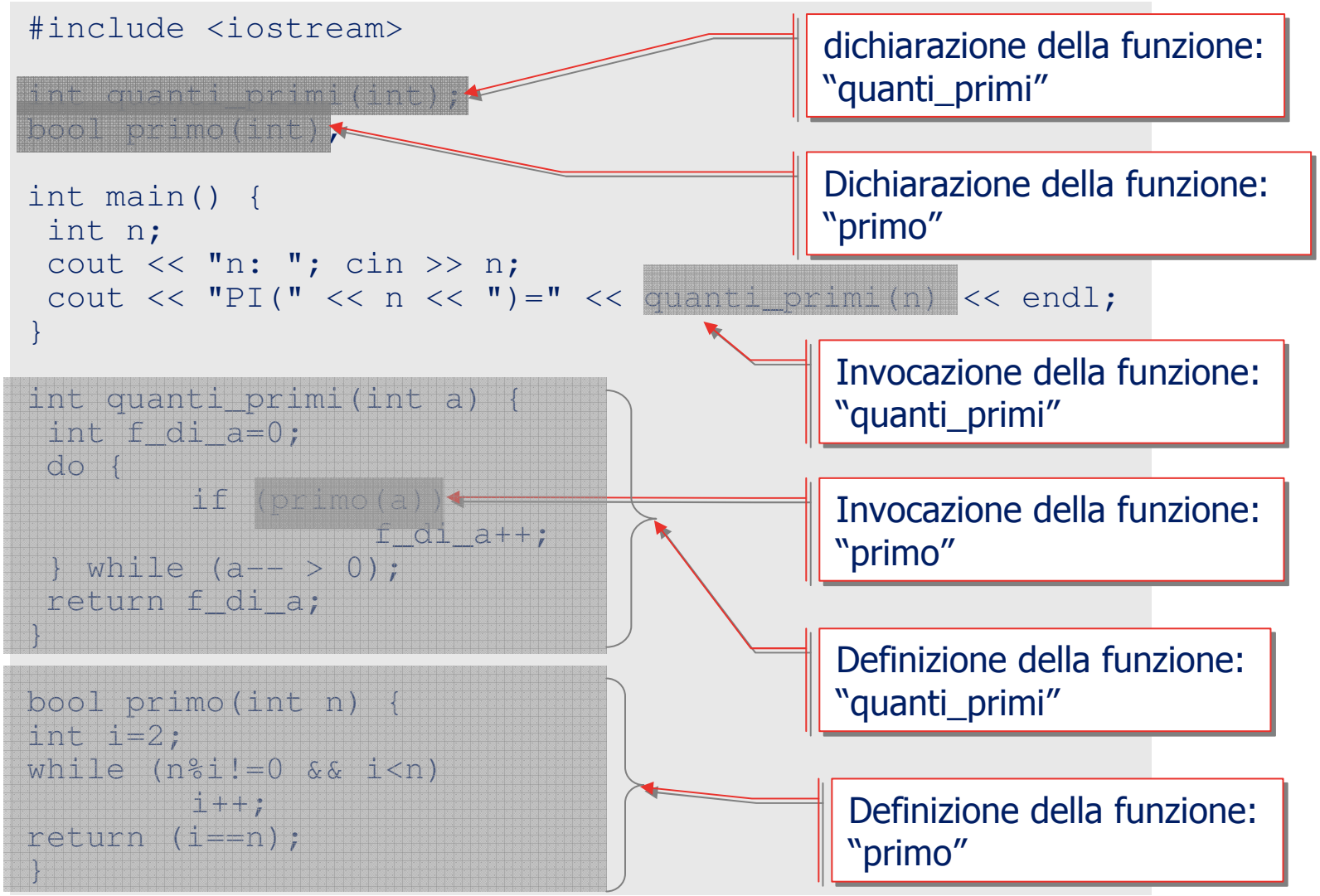
C++: funzioni

- Come l'esempio illustra, prima che nel codice sorgente possa comparire l'**invocazione** di una funzione, tale funzione deve essere stata **dichiarata**, altrimenti il compilatore non ha modo di capire di quale funzione stiamo parlando e verificare che il tipo dei parametri e il tipo del valore di ritorno siano corretti;
- Viceversa, non è necessario che essa sia già stata **definita**, cioè non è necessario che nel sorgente la definizione preceda l'invocazione: la definizione può avvenire in punti successivi del sorgente o addirittura può avvenire *in un altro sorgente*, a patto che della funzione sia data in anticipo la dichiarazione;
- Il motivo è semplice: la dichiarazione specifica al compilatore la **segnatura** o **interfaccia** della funzione, che è tutto e solo ciò che serve per utilizzare quella funzione in maniera corretta; non è invece necessario conoscere la specifica realizzazione della funzione (cioè la sua *definizione*) per scrivere un'invocazione corretta della funzione medesima;
- Si noti che questo è uno dei vantaggi programmatici delle funzioni: *utilizzare del codice senza conoscerne i dettagli, ma solo il significato di alto livello*.

C++: funzioni

- Ogni funzione può - e deve - essere **definita una volta sola**, cioè deve esistere un unico posto nel sorgente in cui una funzione con una certa segnatura è *implementata*;
- Una stessa funzione può essere invece **dichiarata più volte**, quando ne ricorra la necessità. Questo non accade se la funzione viene dichiarata, definita ed utilizzata all'interno di un unico sorgente, ma le cose possono cambiare quando si suddivide il programma in *file sorgenti multipli* o *si utilizzano librerie di funzioni*;
- Una funzione può infine essere **invocata un numero qualunque di volte** e da qualunque punto del sorgente in cui sia visibile la sua dichiarazione; inoltre, **una funzione può invocare a sua volta altre funzioni**, e non c'è limite a questo processo di "nidificazione delle chiamate"; infine una funzione può **invocare se stessa** o - più in generale - anche la funzione che l'ha invocata (direttamente o mediante altre invocazioni intermedie).
- Vedremo più avanti maggiori dettagli sulle funzioni.

C++: funzioni


#include <iostream>
int quanti_primi(int);
bool primo(int);
int main() {
int n;
cout << "n: "; cin >> n;
cout << "PI(" << n << ")=" << quanti_primi(n) << endl;
}
int quanti_primi(int a) {
int f_di_a=0;
do {
if (primo(a))
f_di_a++;
} while (a-- > 0);
return f_di_a;
}
bool primo(int n) {
int i=2;
while (n%i!=0 && i<n)
i++;
return (i==n);
}
dichiarazione della funzione: "quanti_primi"
Dichiarazione della funzione: "primo"
Invocazione della funzione: "quanti_primi"
Invocazione della funzione: "primo"
Definizione della funzione: "quanti_primi"
Definizione della funzione: "primo"

C++: funzioni – variabili locali

- A tempo di esecuzione ogni funzione vive "*in un mondo tutto suo*": per *ogni* distinta invocazione di *ogni* funzione viene creato uno spazio in memoria - chiamato ***record di attivazione*** - in cui la funzione memorizza le sue variabili, dette appunto **variabili locali**;
- Le variabili locali, dunque, nascono con l'invocazione della funzione e **muoiono** (assieme a tutto il record di attivazione) **quando la funzione termina**; inoltre, esse **non sono visibili né modificabili** dall'interno di altre funzioni o dal "**main**" e - al tempo stesso - una funzione ha accesso solo alle proprie variabili locali ma non a quelle delle altre funzioni o a quelle definite nel "**main**";
- I punti in cui una funzione scambia informazioni con l'esterno sono dunque soltanto: **i parametri** - attraverso cui entrano informazioni in *input* dall'esterno (dalla funzione chiamante) - e **il valore di ritorno** - attraverso cui un risultato viene mandato in *output* verso l'esterno (verso la funzione chiamante);
- I parametri (formali) sono in tutto e per tutto delle variabili locali (*non sono visibili dall'esterno e una loro modifica non ha effetti collaterali sull'esterno*), l'unica differenza essendo che i parametri sono **inizializzati automaticamente con i valori passati in ingresso dalla funzione chiamante**.

C++: funzioni – variabili locali e parametri

```
#include <iostream>

int quanti_primi(int);
bool primo(int);

int main() {
 int n;
 cout << "n: "; cin >> n;
 cout << "PI(" << n << ")=" << quanti_primi(n) << endl;
}

int quanti_primi(int a) {
 int f_di_a=0;
 do {
        if (primo(a))
                f_di_a++;
 } while (a-- > 0);
 return f_di_a;
}

bool primo(int n) {
int i=2;
while (n%i!=0 && i<n)
        i++;
return (i==n);
}
```

Variabile "n", locale al "main"

Variabile "f_di_a", locale alla funzione "quanti_primi"

Parametro "a", locale alla funzione "quanti_primi" e inizializzato al valore passato dal chiamante "main"

Parametro "n", locale alla funzione "primo" e inizializzato al valore passato dalla funzione chiamante "quanti_primi"

Variabile "i", locale alla funzione "primo"

C++: funzioni – variabili locali e parametri

```
#include <iostream>

int quanti_primi(int);
bool primo(int);

int main() {
 int n;
 cout << "n: "; cin >> n;
 cout << "PI(" << n << ")="
}

int quanti_primi(int a) {
 int f_di_a=0;
 do {
        if (primo(a))
                f_di_a++;
 } while (a-- > 0);
 return f_di_a;
}

bool primo(int n) {
int i=2;
while (n%i!=0 && i<n)
        i++;
return (i==n);
}
```

Possono esistere in punti diversi del sorgente **variabili o parametri con lo stesso nome**:
essi sono tuttavia oggetti *completamente distinti e indipendenti*, e non interferiscono in nessuna maniera l'uno con l'altro.

Come - ad esempio - queste due variabili "n": esse vivono in distinti momenti spaziali (posizione nel sorgente) e temporali (momento dell'esecuzione in cui esistono e sono accessibili), ed hanno tra l'altro un significato distinto.

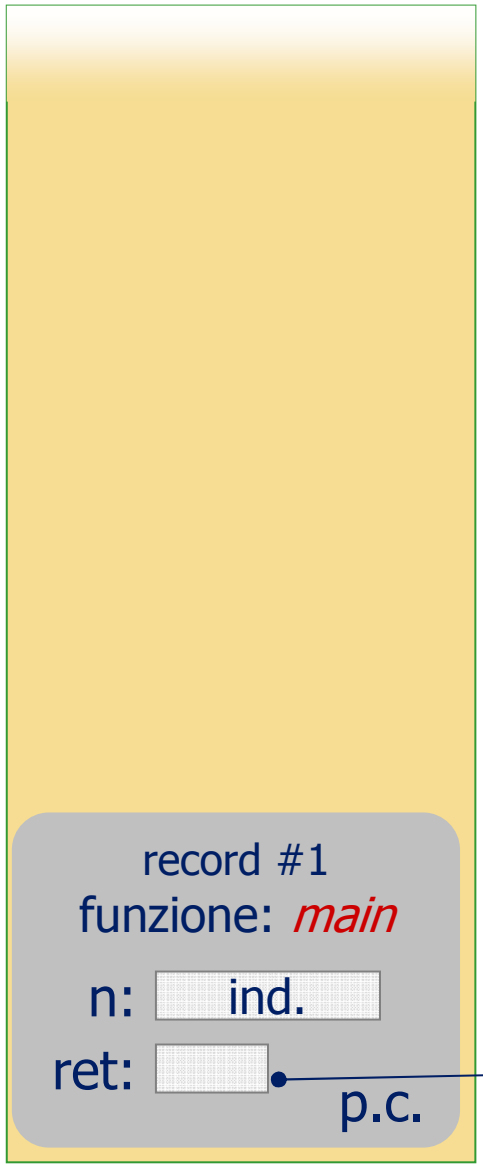
C++: funzioni – variabili locali e parametri

```
#include <iostream>

int quanti_primi(int);
bool primo(int);

int main() {
 int n;
 cout << "n: "; cin >> n;
 cout << "PI(" << n << ")=" << quanti_primi(n) << endl;
}

int quanti_primi(int a) {
 int f_di_a=0;
 do {
        if (primo(a))
                f_di_a++;
 } while (a-- > 0);
 return f_di_a;
}

bool primo(int n) {
int i=2;
while (n%i!=0 && i<n)
        i++;
return (i==n);
}
```

Poiché un parametro non è altro che una variabile locale opportunamente inizializzata, il fatto che il valore di un parametro in una funzione venga modificato non ha alcun effetto sul valore della variabile della funzione chiamante con il cui valore il parametro è stato inizializzato.

In questo caso, ad esempio, mentre il parametro "a" in "quanti_primi" decresce dal suo valore iniziale "n" verso zero man mano che il ciclo viene eseguito, il valore della variabile "n" nel main resta inalterato.

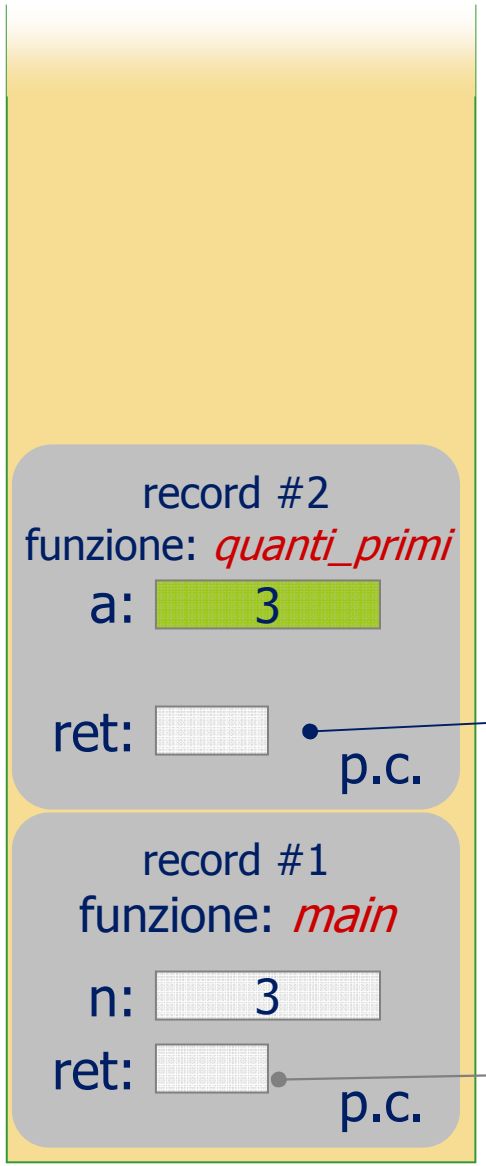
C++: funzioni – stack di attivazione




```
#include <iostream>

int quanti_primi(int);
bool primo(int);

int main() {
 int n;
 cout << "n: "; cin >> n;
 cout << "PI(" << n << ")=";
 cout << quanti_primi(n) << endl;
}

int quanti_primi(int a) {
 int f_di_a=0;
 do {
        if (primo(a))
                f_di_a++;
 } while (a-- > 1);
 return f_di_a;
}

bool primo(int n) {
int i=2;
while (n%i!=0 && i<n)
        i++;
return (i==n);
}
```

Supponiamo che l'utente immetta il valore "3"

C++: funzioni – stack di attivazione

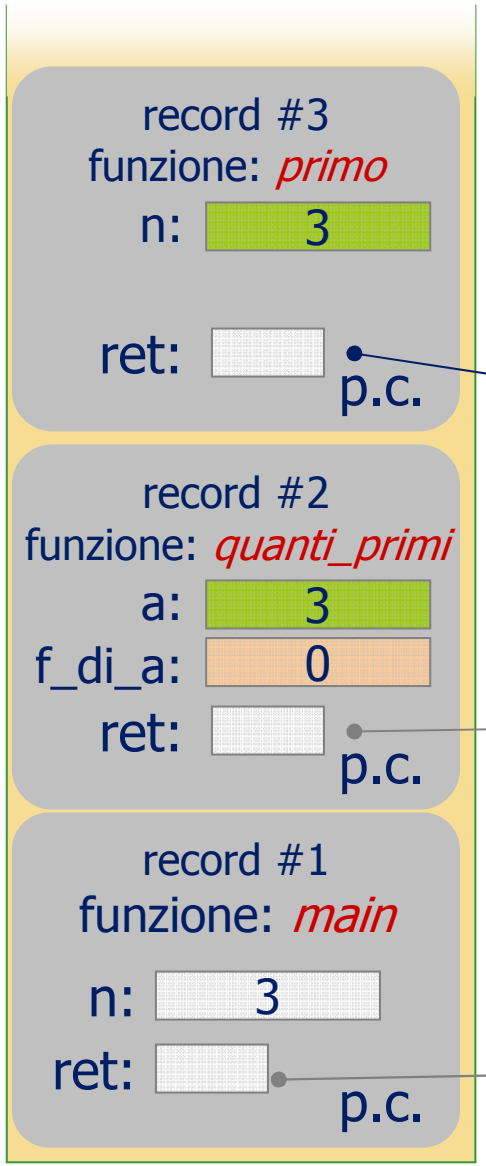



```
#include <iostream>

int quanti_primi(int);
bool primo(int);

int main() {
 int n;
 cout << "n: "; cin >> n;
 cout << "PI(" << n << ")=";
 cout << quanti_primi(n) << endl;
}

int quanti_primi(int a) {
 int f_di_a=0;
 do {
        if (primo(a))
                f_di_a++;
 } while (a-- > 1);
 return f_di_a;
}

bool primo(int n) {
int i=2;
while (n%i!=0 && i<n)
        i++;
return (i==n);
}
```

C++: funzioni – stack di attivazione

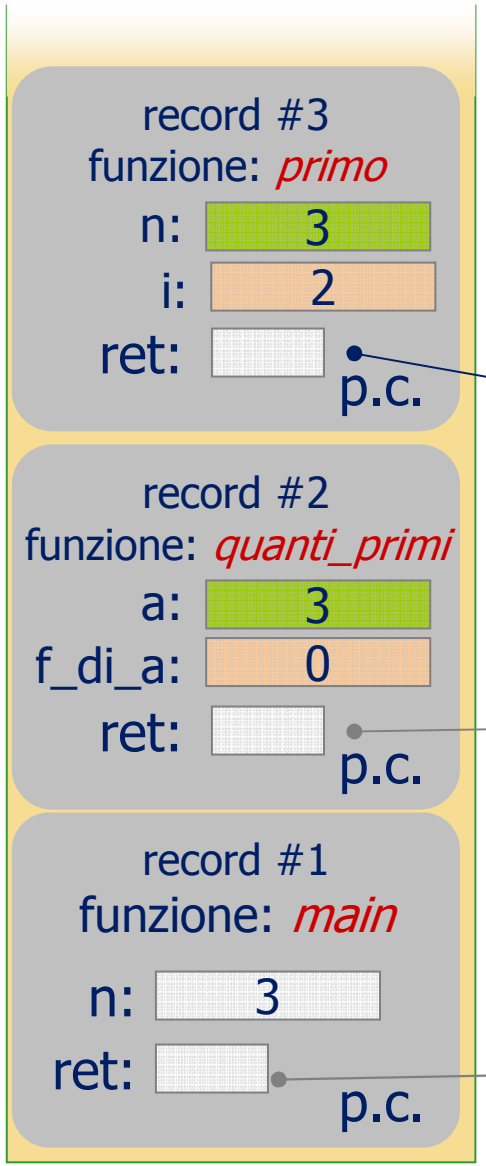

```
#include <iostream>

int quanti_primi(int);
bool primo(int);

int main() {
 int n;
 cout << "n: "; cin >> n;
 cout << "PI(" << n << ")=";
 cout << quanti_primi(n) << endl;
}

int quanti_primi(int a) {
 int f_di_a=0;
 do {
        if (primo(a))
                f_di_a++;
 } while (a-- > 1);
 return f_di_a;
}

bool primo(int n) {
int i=2;
while (n%i!=0 && i<n)
        i++;
return (i==n);
}
```

C++: funzioni – stack di attivazione



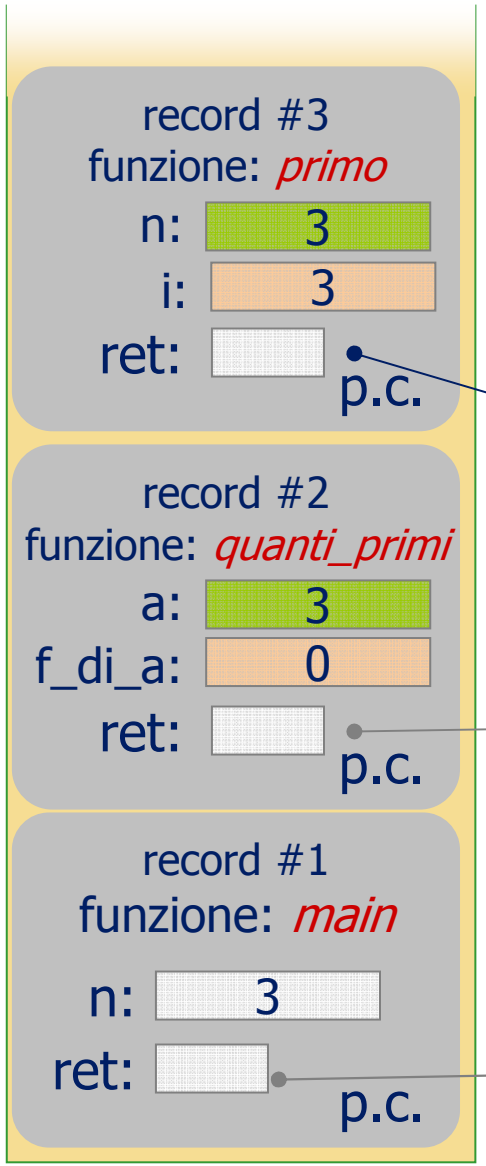

```
#include <iostream>

int quanti_primi(int);
bool primo(int);

int main() {
 int n;
 cout << "n: "; cin >> n;
 cout << "PI(" << n << ")=";
 cout << quanti_primi(n) << endl;
}

int quanti_primi(int a) {
 int f_di_a=0;
 do {
        if (primo(a))
                f_di_a++;
 } while (a-- > 1);
 return f_di_a;
}

bool primo(int n) {
int i=2;
while (n%i!=0 && i<n)
        i++;
return (i==n);
}
```

C++: funzioni – stack di attivazione



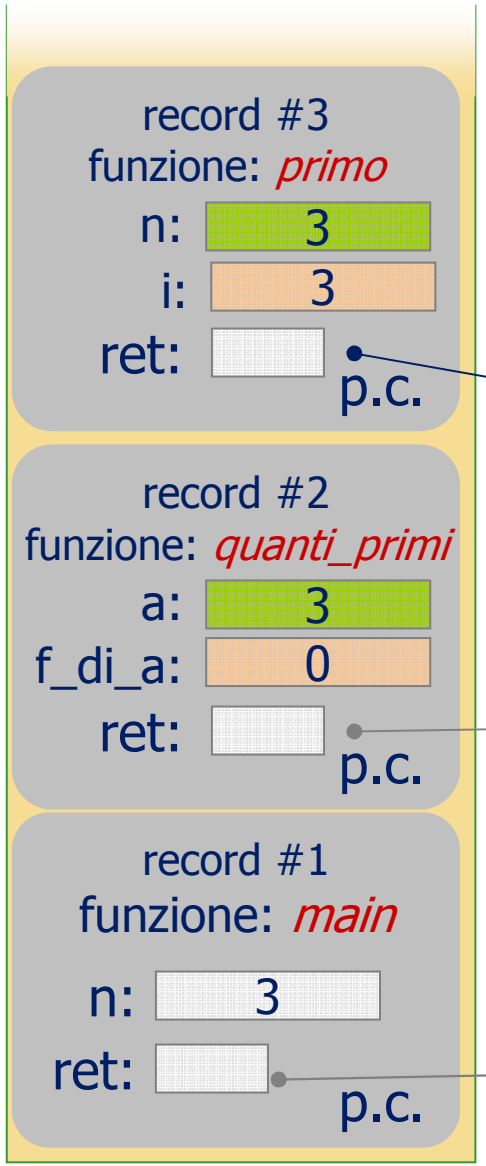

```
#include <iostream>

int quanti_primi(int);
bool primo(int);

int main() {
 int n;
 cout << "n: "; cin >> n;
 cout << "PI(" << n << ")=";
 cout << quanti_primi(n) << endl;
}

int quanti_primi(int a) {
 int f_di_a=0;
 do {
        if (primo(a))
                f_di_a++;
 } while (a-- > 1);
 return f_di_a;
}

bool primo(int n) {
int i=2;
while (n%i!=0 && i<n)
        i++;
return (i==n);
}
```

C++: funzioni – stack di attivazione



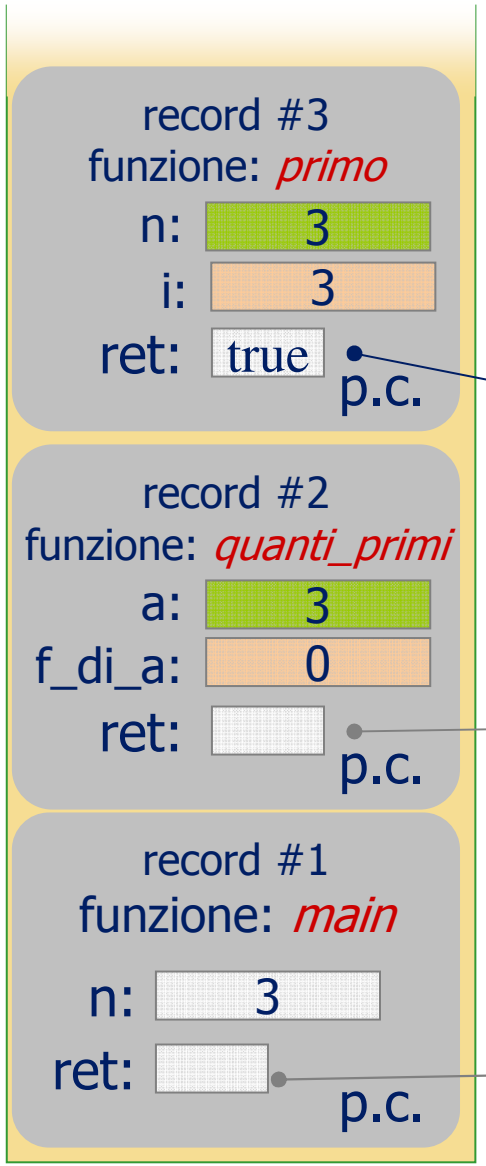

```
#include <iostream>

int quanti_primi(int);
bool primo(int);

int main() {
 int n;
 cout << "n: "; cin >> n;
 cout << "PI(" << n << ")=";
 cout << quanti_primi(n) << endl;
}

int quanti_primi(int a) {
 int f_di_a=0;
 do {
      if (primo(a))
                  f_di_a++;
 } while (a-- > 1);
 return f_di_a;
}

bool primo(int n) {
int i=2;
while (n%i!=0 && i<n)
      i++;
return (i==n);
}
```

C++: funzioni – stack di attivazione




```
#include <iostream>

int quanti_primi(int);
bool primo(int);

int main() {
 int n;
 cout << "n: "; cin >> n;
 cout << "PI(" << n << ")=";
 cout << quanti_primi(n) << endl;
}

int quanti_primi(int a) {
 int f_di_a=0;
 do {
        if (primo(a))
                f_di_a++;
 } while (a-- > 1);
 return f_di_a;
}

bool primo(int n) {
int i=2;
while (n%i!=0 && i<n)
        i++;
return (i==n);
}
```

C++: funzioni – stack di attivazione




```
#include <iostream>

int quanti_primi(int);
bool primo(int);

int main() {
 int n;
 cout << "n: "; cin >> n;
 cout << "PI(" << n << ")=";
 cout << quanti_primi(n) << endl;
}

int quanti_primi(int a) {
 int f_di_a=0;
 do {
        if (primo(a))
                f_di_a++;
 } while (a-- > 1);
 return f_di_a;
}

bool primo(int n) {
int i=2;
while (n%i!=0 && i<n)
        i++;
return (i==n);
}
```

C++: funzioni – stack di attivazione


```
#include <iostream>

int quanti_primi(int);
bool primo(int);

int main() {
 int n;
 cout << "n: "; cin >> n;
 cout << "PI(" << n << ")=";
 cout << quanti_primi(n) << endl;
}

int quanti_primi(int a) {
 int f_di_a=0;
 do {
        if (primo(a))
                f_di_a++;
 } while (a-- > 1);
 return f_di_a;
}

bool primo(int n) {
int i=2;
while (n%i!=0 && i<n)
        i++;
return (i==n);
}
```

C++: funzioni – stack di attivazione


```
#include <iostream>

int quanti_primi(int);
bool primo(int);

int main() {
 int n;
 cout << "n: "; cin >> n;
 cout << "PI(" << n << ")=";
 cout << quanti_primi(n) << endl;
}

int quanti_primi(int a) {
 int f_di_a=0;
 do {
        if (primo(a))
                f_di_a++;
 } while (a-- > 1);
 return f_di_a;
}

bool primo(int n) {
int i=2;
while (n%i!=0 && i<n)
        i++;
return (i==n);
}
```

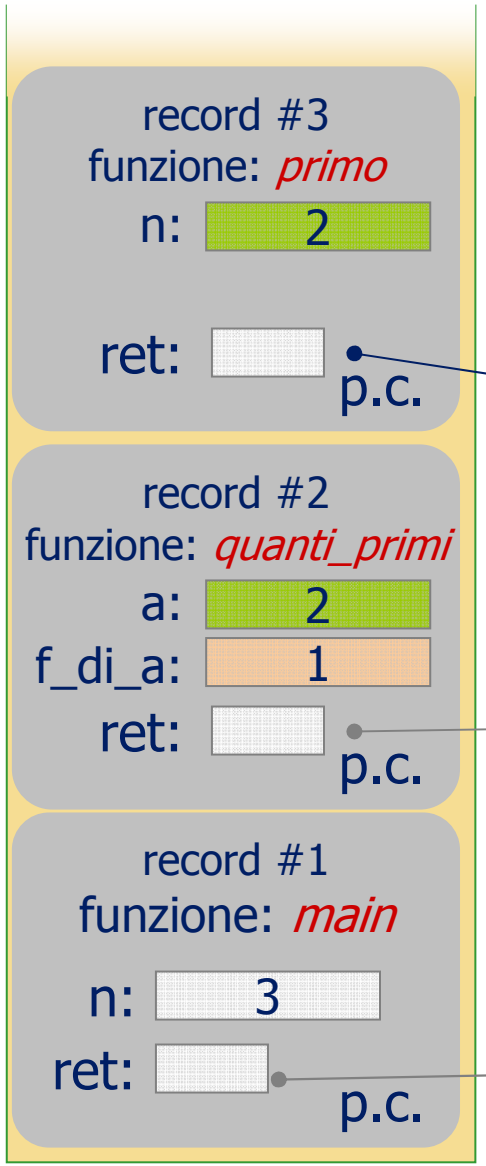
C++: funzioni – stack di attivazione


```
#include <iostream>

int quanti_primi(int);
bool primo(int);

int main() {
 int n;
 cout << "n: "; cin >> n;
 cout << "PI(" << n << ")=";
 cout << quanti_primi(n) << endl;
}

int quanti_primi(int a) {
 int f_di_a=0;
 do {
        if (primo(a))
                f_di_a++;
} while (a-- > 1);
 return f_di_a;
}

bool primo(int n) {
int i=2;
while (n%i!=0 && i<n)
        i++;
return (i==n);
}
```

C++: funzioni – stack di attivazione

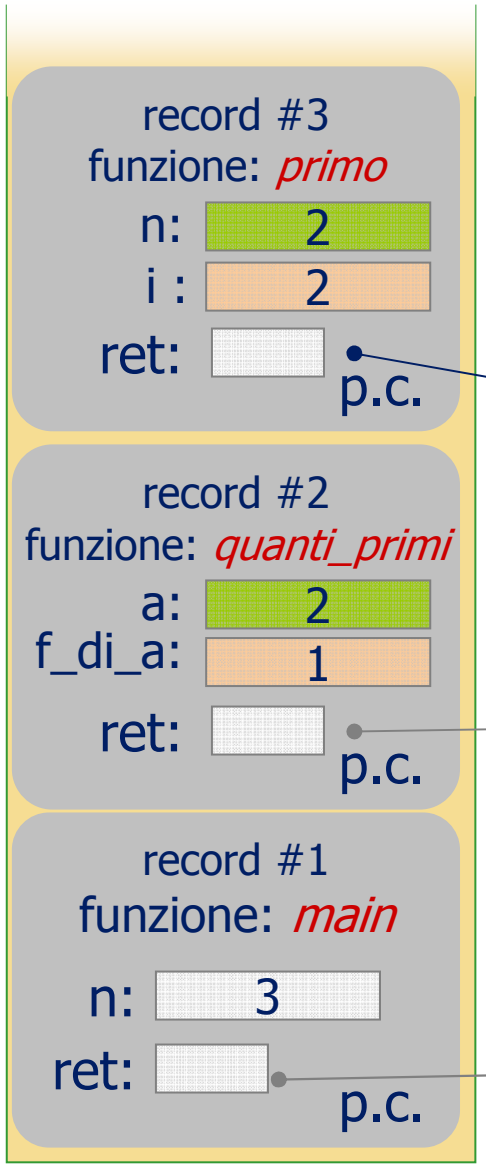

```
#include <iostream>

int quanti_primi(int);
bool primo(int);

int main() {
 int n;
 cout << "n: "; cin >> n;
 cout << "PI(" << n << ")=";
 cout << quanti_primi(n) << endl;
}

int quanti_primi(int a) {
 int f_di_a=0;
do {
        if (primo(a))
                f_di_a++;
 } while (a-- > 1);
 return f_di_a;
}

bool primo(int n) {
int i=2;
while (n%i!=0 && i<n)
        i++;
return (i==n);
}
```

C++: funzioni – stack di attivazione



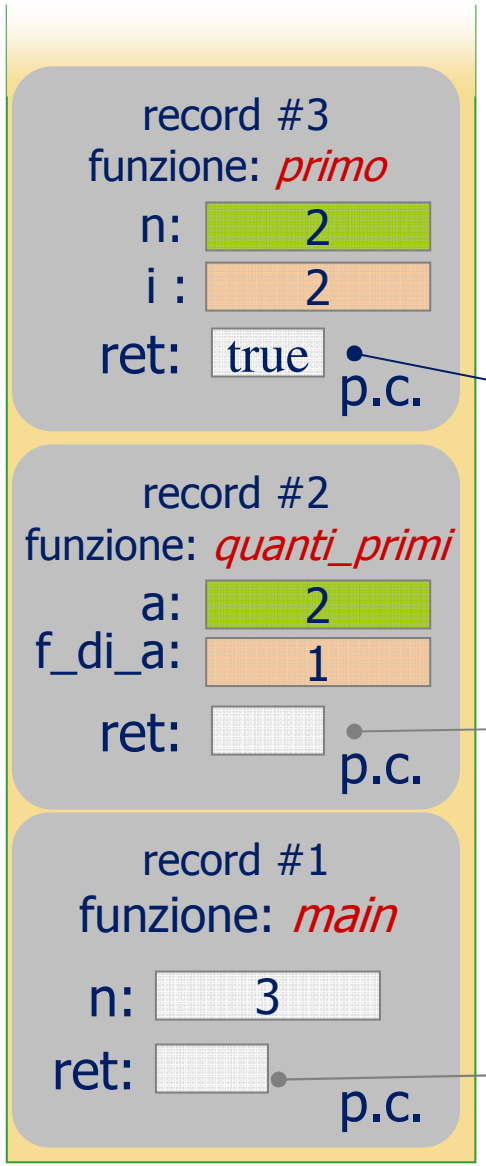

```
#include <iostream>

int quanti_primi(int);
bool primo(int);

int main() {
 int n;
 cout << "n: "; cin >> n;
 cout << "PI(" << n << ")=";
 cout << quanti_primi(n) << endl;
}

int quanti_primi(int a) {
 int f_di_a=0;
 do {
        if (primo(a))
                f_di_a++;
 } while (a-- > 1);
 return f_di_a;
}

bool primo(int n) {
int i=2;
while (n%i!=0 && i<n)
        i++;
return (i==n);
}
```

C++: funzioni – stack di attivazione



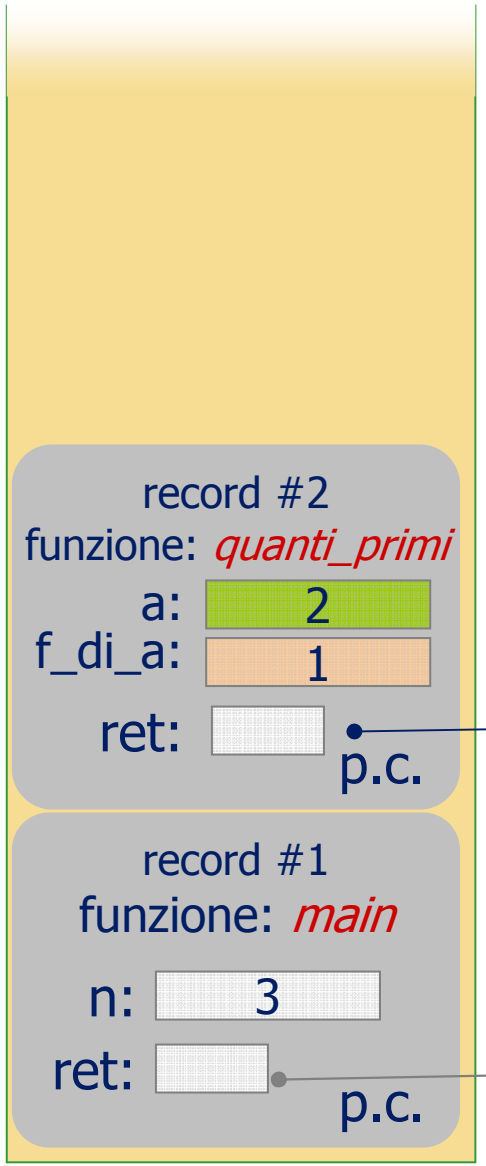

```
#include <iostream>

int quanti_primi(int);
bool primo(int);

int main() {
 int n;
 cout << "n: "; cin >> n;
 cout << "PI(" << n << ")=";
 cout << quanti_primi(n) << endl;
}

int quanti_primi(int a) {
 int f_di_a=0;
 do {
        if (primo(a))
                f_di_a++;
 } while (a-- > 1);
 return f_di_a;
}

bool primo(int n) {
int i=2;
while (n%i!=0 && i<n)
        i++;
return (i==n);
}
```

C++: funzioni – stack di attivazione




```
#include <iostream>

int quanti_primi(int);
bool primo(int);

int main() {
 int n;
 cout << "n: "; cin >> n;
 cout << "PI(" << n << ")=";
 cout << quanti_primi(n) << endl;
}

int quanti_primi(int a) {
 int f_di_a=0;
 do {
        if (primo(a))
                f_di_a++;
 } while (a-- > 1);
 return f_di_a;
}

bool primo(int n) {
int i=2;
while (n%i!=0 && i<n)
        i++;
return (i==n);
}
```

C++: funzioni – stack di attivazione

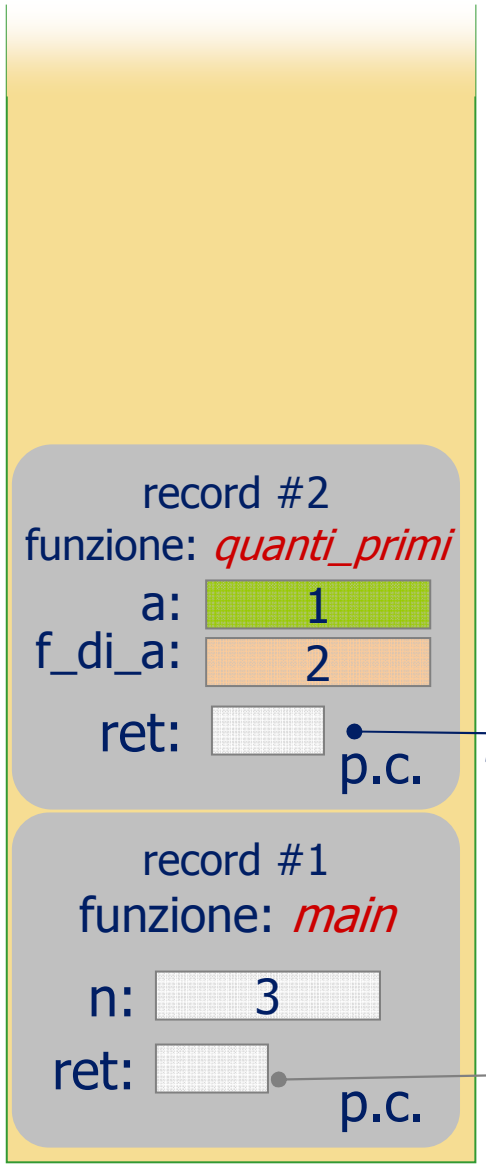



```
#include <iostream>

int quanti_primi(int);
bool primo(int);

int main() {
 int n;
 cout << "n: "; cin >> n;
 cout << "PI(" << n << ")=";
 cout << quanti_primi(n) << endl;
}

int quanti_primi(int a) {
 int f_di_a=0;
 do {
        if (primo(a))
                f_di_a++;
 } while (a-- > 1);
 return f_di_a;
}

bool primo(int n) {
int i=2;
while (n%i!=0 && i<n)
        i++;
return (i==n);
}
```

C++: funzioni – stack di attivazione


```
#include <iostream>

int quanti_primi(int);
bool primo(int);

int main() {
 int n;
 cout << "n: "; cin >> n;
 cout << "PI(" << n << ")=";
 cout << quanti_primi(n) << endl;
}

int quanti_primi(int a) {
 int f_di_a=0;
 do {
        if (primo(a))
                f_di_a++;
 } while (a-- > 1);
 return f_di_a;
}

bool primo(int n) {
int i=2;
while (n%i!=0 && i<n)
        i++;
return (i==n);
}
```

C++: funzioni – stack di attivazione

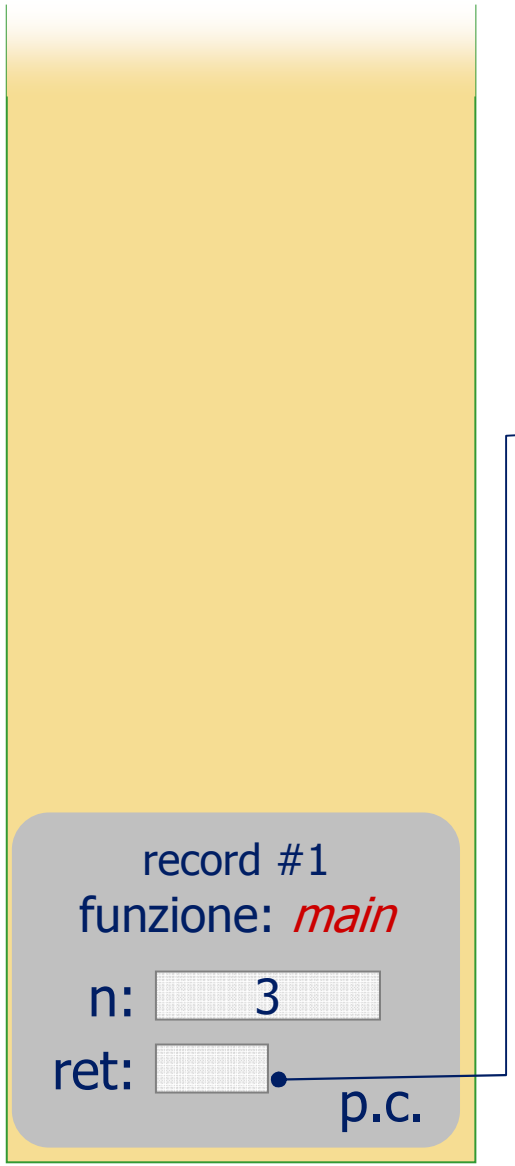



```
#include <iostream>

int quanti_primi(int);
bool primo(int);

int main() {
 int n;
 cout << "n: "; cin >> n;
 cout << "PI(" << n << ")=";
 cout << quanti_primi(n) << endl;
}

int quanti_primi(int a) {
 int f_di_a=0;
 do {
        if (primo(a))
                f_di_a++;
 } while (a-- > 1);
 return f_di_a;
}

bool primo(int n) {
int i=2;
while (n%i!=0 && i<n)
        i++;
return (i==n);
}
```

C++: funzioni – stack di attivazione


```
#include <iostream>

int quanti_primi(int);
bool primo(int);

int main() {
 int n;
 cout << "n: "; cin >> n;
 cout << "PI(" << n << ")=";
 cout << quanti_primi(n) << endl;
}

int quanti_primi(int a) {
 int f_di_a=0;
 do {
        if (primo(a))
                f_di_a++;
 } while (a-- > 1);
 return f_di_a;
}

bool primo(int n) {
int i=2;
while (n%i!=0 && i<n)
        i++;
return (i==n);
}
```

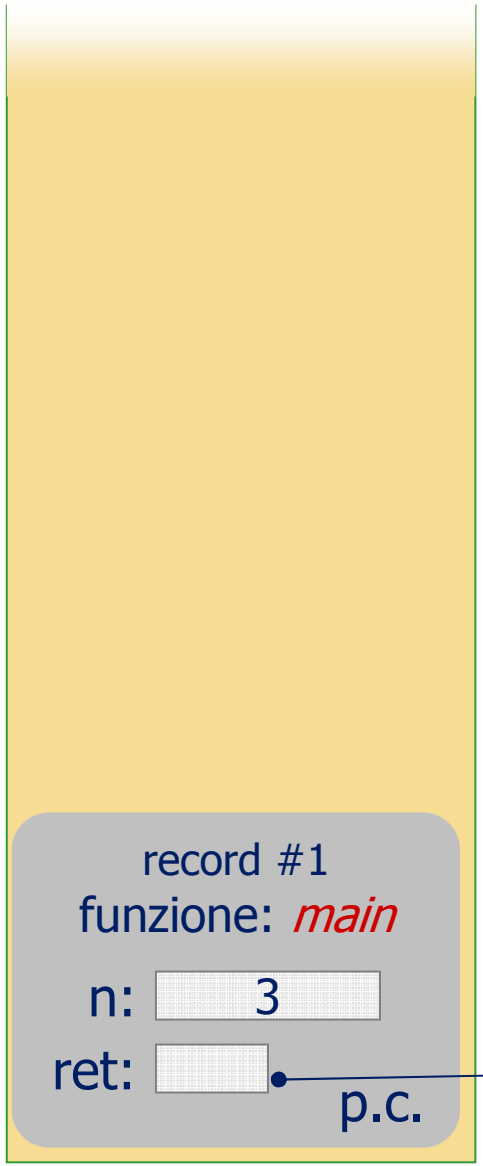
C++: funzioni – stack di attivazione




```
#include <iostream>

int quanti_primi(int);
bool primo(int);

int main() {
 int n;
 cout << "n: "; cin >> n;
 cout << "PI(" << n << ")=";
 cout << quanti_primi(n) << endl;
}

int quanti_primi(int a) {
 int f_di_a=0;
 do {
        if (primo(a))
                f_di_a++;
 } while (a-- > 1);
 return f_di_a;
}

bool primo(int n) {
int i=2;
while (n%i!=0 && i<n)
        i++;
return (i==n);
}
```

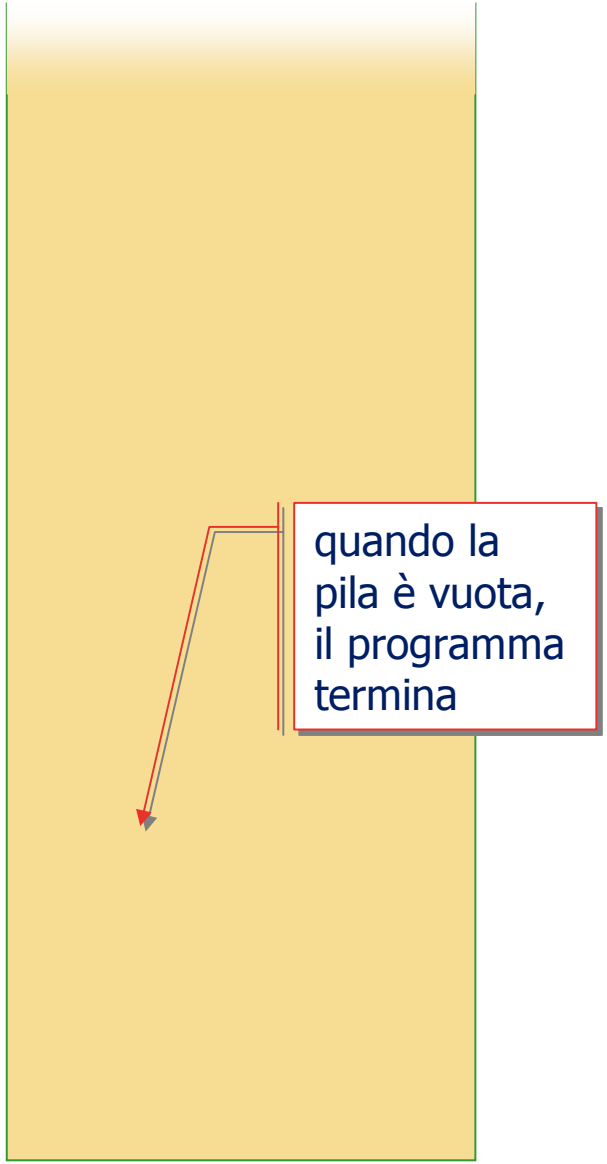
viene stampato:
`PI(3)=2`

C++: funzioni – stack di attivazione




```
#include <iostream>

int quanti_primi(int);
bool primo(int);

int main() {
 int n;
 cout << "n: "; cin >> n;
 cout << "PI(" << n << ")=";
 cout << quanti_primi(n) << endl;
}

int quanti_primi(int a) {
 int f_di_a=0;
 do {
        if (primo(a))
                f_di_a++;
 } while (a-- > 1);
 return f_di_a;
}

bool primo(int n) {
int i=2;
while (n%i!=0 && i<n)
        i++;
return (i==n);
}
```

C++: funzioni – stack di attivazione




```
#include <iostream>

int quanti_primi(int);
bool primo(int);

int main() {
 int n;
 cout << "n: "; cin >> n;
 cout << "PI(" << n << ")=";
 cout << quanti_primi(n) << endl;
}

int quanti_primi(int a) {
 int f_di_a=0;
 do {
        if (primo(a))
                f_di_a++;
 } while (a-- > 1);
 return f_di_a;
}

bool primo(int n) {
int i=2;
while (n%i!=0 && i<n)
        i++;
return (i==n);
}
```

C++: funzioni ricorsive

- Dal punto di vista sintattico, siamo in presenza di una funzione ricorsiva quando **all'interno della definizione di una funzione compaiono una o più invocazioni alla funzione che si sta definendo**; in altre parole, la funzione "ricorre a se stessa" per svolgere il proprio compito;
- Questo tipo di ricorsione è chiamata *ricorsione diretta*; esistono casi più complessi, come quello in cui nel corpo della funzione f1 si invoca la funzione f2 e al tempo stesso nel corpo di f2 si invoca la funzione f1 (si parla in questo caso di *mutua ricorsione*); esistono tipi di ricorsione ancora più generali, ma non li vedremo all'interno del corso;
- Le funzioni ricorsive non sono -come potrebbe sembrare- condannate a "non terminare mai", perchè l'invocazione ricorsiva (o, brevemente: *ricorsione*) non avviene *incondizionatamente*; ci *devono essere*, al contrario, dei casi in cui la funzione spezza la catena della ricorsione e riesce a calcolare il risultato che le è richiesto senza ulteriori invocazioni ricorsive;
- *Le funzioni **ricorsive** risultano estremamente comode nella codifica di tutti gli algoritmi modellati su un procedimento di soluzione **induttivo**.*

C++: funzioni ricorsive

- Una volta descritto in maniera induttiva il procedimento di soluzione di un problema, la sua implementazione ricorsiva in C++ ricalca in genere il seguente schema:

```
SOLUZIONE funzione_ricorsiva (PROBLEMA p, altri parametri...) {

 if (siamo nel caso base) {

      risolvi direttamente il problema p: sia "sol" la soluzione;
      return sol;

 } else { //siamo nel caso induttivo

      estrai da p uno o più sottoproblemi p_i di dimensione minore di p;

      per ogni i, sia:
            s_i = funzione_ricorsiva(p_i, altri parametri...);

      componi le soluzioni s_i ed effettua gli altri calcoli necessari
      ad ottenere una soluzione "sol" del problema p;
      return sol;
 }
}
```

C++: funzioni ricorsive

- Dallo schema descritto si intuisce come le funzioni ricorsive possano garantire la terminazione: ogni volta che una funzione invoca se stessa, sta richiedendo la soluzione di un problema *più semplice* di quello di partenza; prima o poi si arriverà dunque ad un problema di dimensione sufficientemente piccola da poter essere affrontato nel *caso base*; a questo punto viene restituita una risposta che, a cascata e all'indietro, permette di calcolare le risposte intermedie più complesse rimaste "in sospeso";
- Nello schema indicato restano da dettagliare caso per caso una serie di elementi:
  - il **caso base:** può essere esso stesso complesso da risolvere, possono esistere più casi base distinti, ecc.
  - il **caso induttivo** comprende due fasi fondamentali da progettare di volta in volta:
    - la fase dell'estrapolazione del/dei sottoproblema/i;
    - la fase dello sfruttamento della/delle soluzione/i parziale/i;

  in genere una di queste fasi (quale delle due dipende dal procedimento di soluzione) risulta di complessità predominante rispetto all'altra.

C++: funzioni ricorsive

- Una esempio molto semplice di problema la cui soluzione si presta ad essere descritta in maniera induttiva è il calcolo del fattoriale n! di un intero n:

$$n! = \prod_{i=1}^{n} i = 1 \cdot 2 \cdot \cdots \cdot n \quad (0! = 1)$$

- La descrizione induttiva del calcolo è:
  caso base: per n=0 si ha *direttamente* n! = 1
  caso induttivo: per n>0 si ha n! = n * (n-1)!
- In questo esempio: il *caso base* è semplice da riconoscere e calcolare, la *ricorsione* riguarda un solo sottoproblema la cui estrazione è semplice (si passa dal valore "n" al valore "n-1") e lo *sfruttamento della soluzione parziale* è operata tramite la moltiplicazione;
- L'implementazione è dunque:

```
int fattoriale (int n) {
  if (n==0)
        return 1;
  else
        return n * fattoriale(n-1);
}
```

C++: I numeri di Fibonacci

- La formula per calcolare i numeri di Fibonacci è dunque:

$$fib(n) = \begin{cases} 1 & \text{se } n = 0 \text{ oppure } n = 1 \\ fib(n-1) + fib(n-2) & \text{se } n > 1 \end{cases}$$

- In questo esempio: il *caso base* è semplice da riconoscere e calcolare, la *ricorsione* riguarda due sottoproblemi di semplice individuazione (si passa dal valore "n" ai valori "n-1" e "n-2"); lo *sfruttamento della soluzione parziale* è operato tramite la semplice *addizione* delle soluzioni dei sottoproblemi individuati;
- L'implementazione è dunque (tre versioni equivalenti):

```
int fib(int n) {
 if (n==0 || n==1)
   return 1;
 else
   return fib(n-1)+fib(n-2);
}
```

```
int fib (int n) {
 int ris;
  if (n==0 || n==1)
    ris = 1;
  else
    ris = fib(n-1)+fib(n-2);
 return ris;
}
```

```
int fib(int n) {
  return (n==0||n==1)? 1 : (fib(n-1)+fib(n-2));
}
```

C++: funzioni ricorsive vs iterative

- Vantaggio principale:
  - Intuitive per definire particolari tipi di funzioni che sono naturalmente ricorsive.
- Svantaggio principale:
  - Molto spesso le versioni ricorsive possono risultare poco efficienti. Ogni invocazioni richiede la allocazione di un record di attivazione.
  - Siccome l'area dati dedicata allo stack di attivazione è limitato, può accadere che si presenti il cosiddetto problema dello *stack overflow* (superamento della dimensione dello stack).
  - Per sopperire a questi problemi alcuni compilatori automaticamente trasformano funzioni ricorsive in equivalenti funzioni iterative, in cui si richiede un solo record di attivazione.

C++: funzioni ricorsive vs iterative

- Il fattoriale iterativo diventa:

```
int fattoriale(int N) {
  int r = 1;
  for( ; N > 1; N--)
    r = r * N;
  return r;
}
```

C++: funzioni ricorsive vs iterative

- Il calcolo della funzione di fibonacci iterativa?

```
int fibonacci(int N) {
  int F[N+1];
  F[0] = 1;
  F[1] = 1;
  for(int i = 2; i <= N; i++)
     F[i] = F[i-1] + F[i-2];
  return F[N];
}
```

C++: funzioni e side effects

- Il modo di utilizzare parametri e funzioni che abbiamo appena visto si chiama “puramente funzionale”, con ciò si intende che una funzione C++ viene vista come se fosse una funzione nel senso matematico: a determinati valori dell’input, viene associato un valore di ritorno, e non ci sono effetti collaterali sui parametri del chiamante o su altri oggetti globali; in altre parole, dopo che una funzione è stata eseguita, il suo valore di ritorno è l’unica traccia lasciata, l’unico testimone dell’esecuzione (oltre al tempo speso dalla funzione per effettuare i suoi calcoli): null’altro è stato affetto da tale esecuzione;
- Tuttavia, spesso può essere comodo o necessario che una funzione procuri degli *effetti collaterali*: sui parametri di invocazione, su oggetti globali o su altro ancora (ad esempio: possiamo volere che stampi qualcosa sullo schermo). A volte può addirittura darsi l’evenienza che ad una funzione *non* si richieda di ritornare un risultato, ma ci interessino *solo* i suoi effetti collaterali;
- Queste considerazioni ci portano ad introdurre due nuovi argomenti:
  - le funzioni che non ritornano alcun valore (conosciute anche come **procedure**);
  - le funzioni che modificano il valore dei parametri del chiamante.

C++: procedure

- Nella dichiarazione (e nella definizione) di una funzione si può usare la parola chiave "**void**" come tipo di ritorno per indicare che la funzione in questione non ritorna alcun valore;
- Si parla in questo casi di **procedura** invece che di funzione; una procedura utilizza ancora la parola chiave "**return**" per terminare, ma senza alcun valore a seguire; se invece l'istruzione "**return**" non è presente, la funzione ritorna al momento del suo "termine naturale", cioè quando si arriva all'ultima graffa chiusa nel codice;
- Ad esempio la funzione:

```
void stampa(char c,int i) {
for (int j=0;j<i;j++)
        cout << c;
return;
}
```

  … stampa "i" volte il carattere "c" sullo schermo, e questo è il suo unico scopo: non c'è valore di ritorno, e infatti l'istruzione "**return**" non è seguita da alcuna espressione (in questo caso specifico poteva anche essere omessa);
- Una chiamata a procedura **non** può dunque apparire in un'espressione, ma al contrario compare nel codice del chiamante come *un'istruzione*.

C++: passaggio per valore o per riferimento

- Il tipo di passaggio di parametri che abbiamo usato fin'ora - caratterizzato dal fatto che nel corpo delle funzioni i parametri formali sono assimilabili a variabili locali inizializzate al valore dei corrispondenti parametri attuali del chiamante e dal conseguente fatto che il valore di tali parametri nel chiamante non è in alcun modo affetto dall'esecuzione della funzione chiamata - si chiama **passaggio per valore**: questo nome mette in evidenza che è il *valore* ad essere passato e *non la variabile che lo contiene*;
- Il C++ supporta in aggiunta il così detto **passaggio per riferimento**: quando un parametro è passato per riferimento, il record di attivazione non contiene una copia locale del suo valore; al contrario, ci si *riferisce* direttamente alla (memoria contenente la) variabile nel corpo del chiamante, per cui qualunque variazione apportata nel corpo di una funzione al valore dei parametri in ingresso passati per riferimento si riflette sul valore delle relative variabili nella funzione chiamante;
- Per indicare al compilatore che un parametro deve essere passato ad una funzione per riferimento si inserisce una "&" dopo il nome del tipo del parametro in questione sia nella dichiarazione che nella definizione della funzione, mentre la forma sintattica dell'invocazione resta invariata.

C++: passaggio per valore

- Supponiamo ad esempio di voler scrivere una funzione “scambia” che non ha un valore di ritorno, ma che inverte i valori dei suoi due parametri interi in ingresso; la definizione:

```
void scambia(int a, int b) {
        int temp = a;
        a = b;    b = temp;
        return;
}
```

pur sintatticamente corretta, non ha l’effetto sperato: i parametri sono passati *per valore*, sicchè un programma che la invochi in questo modo:

```
int main() {
  int v1=3; int v2=5;
  scambia (v1, v2);
  cout << v1 << v2;
}
```

stamperebbe prima 3 e poi 5 (esattamente come se la funzione “scambia” non fosse stata invocata).

C++: passaggio per riferimento

- Se invece specifichiamo che i parametri vengano passati per riferimento:

```
void scambia(int &a, int &b) {
        int temp = a;
        a = b;     b = temp;
        return;
}
```

abbiamo definito una funzione che, se invocata in questo modo:

```
int main() {
  int v1=3; int v2=5;
  scambia (v1,v2);
  cout << v1 << v2;
}
```

causa effettivamente lo scambio dei valori delle variabili v1 e v2 del main, per modo che si ottiene la stampa del valore 5 e poi del valore 3.

C++: passaggio per riferimento

- Il passaggio per riferimento è utile nelle seguenti circostanze:
  - Quando si vuole che la funzione effettui side effect sui parametri di invocazione (come nel caso della funzione “scambia”);
  - Quando si vogliono utilizzare i parametri non solo per l’ingresso di informazioni in una funzione, ma anche per l’uscita: ad esempio, si vuole scrivere una funzione che ritorni *due* valori al chiamante;
  - Quando l’oggetto da passare ad una funzione è così grande (può accadere nel caso esso sia un oggetto istanza di una classe) che considerazioni d’efficienza spingano a non voler affrontare l’onere (di tempo e di memoria) necessario a produrne una copia;
- Alcuni *potenziali* svantaggi del passaggio per riferimento sono tuttavia:
  - Si perde la *pulizia concettuale* dello stile puramente funzionale;
  - Non è più evidente leggendo il codice *chi e dove* può alterare il valore di una variabile;
  - Gli effetti collaterali sui parametri possono essere *involontari* e causare errori;
  - *Più identificatori distinti* possono riferirsi allo *stesso oggetto*;
  - Il parametro attuale passato deve avere un *left-value* (deve essere una variabile e non - ad esempio - il risultato della valutazione di un’espressione).

C++: passaggio di array

- Un discorso a parte merita il *passaggio di array* tra funzioni:
  - Una funzione non può avere come tipo di ritorno un array;
  - Quando si passa un array ad una funzione, si sta *implicitamente* passando il contenuto dell'array per *riferimento* e dunque la funzione *può modificarlo* (questo comportamento è attuato per compatibilità con il linguaggio C, nel quale un array è in realtà un *puntatore* che permette comunque la modifica degli argomenti cui punta anche se il valore del puntatore medesimo è passato per valore);
  - La dichirazione di una funzione "funz" che accetta un array (monodimensionale) di interi è ad esempio:
    TIPO_RITORNO funz(int []);

    mentre la sua definizione sarà del tipo:
    TIPO_RITORNO funz(int a[]) { … }
  - Non è dunque necessario specificare la dimensione dell'array nel tipo di parametro, ma se si vuole che la funzione invocata conosca tale dimensione bisogna premurarsi di passarla esplicitamente:
    TIPO_RITORNO funz(int a[], int dim) { … }
  - Quando si passano array multidimensionali, la dimensione deve invece essere data esplicitamente nel tipo del parametro:
    TIPO_RITORNO funz(bool [10][10]);
    TIPO_RITORNO funz(bool matrice[10][10]) { … }

C++: passaggio di array

```
#include <iostream>

using namespace std;

void riempi(int a[], int dim) {
    for (int i=0; i<dim; i++) a[i]=i;
}

void stampa(int a[], int dim) {
    for (int i=0; i<dim; i++)  cout << a[i];
}

int somma(int a[], int dim) {
   int tot=0;
   for (int i=0; i<dim; i++)  tot +=a[i];
   return tot;
}

int main () {
   int val[10];
   riempi(val,10);
   stampa (val,10);
   cout << endl << somma (val,10) << endl;
}
```

questa funzione riempie l'array in ingresso di valori crescenti in maniera che ogni cella contenga un valore pari al proprio indice

questa funzione stampa sullo schermo il contenuto dell'array che le viene passato

questa invocazione stamperà sullo schermo:

```
0123456789
```

questa istruzione stamperà sullo schermo il valore:

```
45
```

C++: la funzione main

- Anche il "**main**" è una funzione (con alcune caratteristiche particolari):
  - Il fatto che non sia specificato il tipo di ritorno significa che è sottinteso che la funzione ritorna un intero; anche se quasi tutti i compilatori accettano questo sottinteso, è buona norma esplicitare sempre il tipo di ritorno:
    int main() {…}
  - Dato che il **main** ritorna un **int**, bisognerebbe terminare tale funzione con un'istruzione "**return**" che ritorni un intero *a chi ha invocato il programma*; tale intero viene per convenzione utilizzato per distinguere esecuzioni che terminano senza errori (valore di ritorno: zero) da esecuzioni che terminano con qualche errore (valore di ritorno: diverso da zero; il valore assoluto dell'intero ritornato è in genere un codice che identifica il tipo di errore occorso); (solo) la funzione **main** può esimersi dal ritornare esplicitamente un valore (come abbiamo fatto fin'ora): in questo caso il valore di ritorno sottinteso è zero;
  - La funzione **main** può essere definita come se fosse dichiarata senza parametri (come fatto fin'ora):

    int main() {…}
  - La funzione **main** ha però anche una segnatura alternativa:

    int main(int argc, **char** *argv[]) {…}

    che utilizza i puntatori (che vedremo più avanti nel corso); questa forma serve per accettare in ingresso dei parametri dalla linea di comando.

C++: funzioni di libreria

- Una ***libreria*** è un insieme di funzioni pre-compilate. Alcune librerie sono disponibili in tutte le implementazioni.
- Una libreria è formata da una coppia di file:
  - Un file di ***intestazione*** (***header***) contenente le ***dichiarazioni*** delle funzioni definite nella libreria.
  - Un file contenente le *funzioni compilate*.
- Per utilizzare una libreria bisogna:
  - Includere il file di intestazione della libreria con la direttiva:

    #include <nomelibreria.h>

    in alcuni casi il ".h" può essere omesso.
  - Indicare al linker il file contenente le funzioni compilate della libreria.
  - Riprenderemo più avanti questo concetto.

C++: libreria matematica

- Ereditata dal C

  #include <cmath>

  - **`fabs(x)`** valore assoluto di tipo float
  - **`sqrt(x)`** radice quadrata di x
  - **`pow(x,y)`** eleva x alla potenza di y (x^y)
  - **`exp(x)`** eleva **e** alla potenza di y ($\mathbf{e}^x$)
  - **`log(x)`** logaritmo naturale di x
  - **`log10(x)`** logaritmo in base 10 di x
  - **`sin(x)`** e **`asin(x)`** seno e arcoseno trigonometrico
  - **`cos(x)`** e **`asin(x)`** coseno e arcoseno trigonom.
  - **`tan(x)`** e **`atan(x)`** tangente e arcotangente trig.

C++: input output

- Un programma comunica con l'esterno tramite uno o più ***flussi*** (***stream***).
- Uno ***stream*** è una struttura logica costituita da una ***sequenza di caratteri***, in numero teoricamente infinito, terminante con un apposito carattere che ne identifica la fine.
- Gli ***stream*** vengono associati (con opportuni comandi) ai dispositivi fisici collegati al computer (tastiera, video, stampante) o a file residenti sulla memoria di massa.

C++: input output

- In C++ esistono i seguenti **stream predefiniti**
  - **`cin`** (stream standard di ingresso)
  - **`cout`** (stream standard di uscita)
  - **`cerr`** (stream standard di errore)

- Le funzioni che operano su questi stream sono in una libreria di ingresso/uscita e per usarle occorre la direttiva

```
#include <iostream>
```

C++: input

- Lo stream di ingresso standard (**`cin`**) è quello da cui il programma preleva i dati ed è tipicamente associato alla tastiera
- Per prelevare dati si usa l'istruzione di lettura
  `stream >> variabile;`
- L'istruzione di lettura comporta
  - il prelievo dallo stream di una sequenza di caratteri
  - la conversione di tale sequenza di caratteri in un valore che viene assegnato alla variabile
- Con un'istruzione del tipo **`cin >> x;`**
- se **`x`** è di tipo **`char`** e lo stream contiene
  ……□□□□a□−14.53□□728……
  ↑ ⟶
- se il carattere selezionato dal puntatore (↑)
  non è una spaziatura (□)
  - il carattere viene prelevato
  - il carattere viene assegnato alla variabile x
  - il puntatore si sposta alla casella successiva
- altrimenti il puntatore si sposta in avanti per trovare un carattere che non sia una spaziatura

C++: input

- Con un'istruzione del tipo **`cin >> x;`**
- se **`x`** è di tipo **`int`** e lo stream contiene
  `.....□□□□-157□□□Ciao□□6.28□.....`
  ↑ ⟶
- la variabile **`x`** conterrà il valore **`-157`** e il puntatore si sarà spostato fino al carattere seguente la cifra **`7`**
- Con uno stream che invece contiene
  `.....□□□Ciao□□□□-157□□.....`
  ↑ ⟶
  l'operazione di prelievo non avviene e lo stream si porta in uno stato di errore
- L'istruzione di ingresso ha una forma più generale che consente **letture multiple**
- Ad esempio, con un'istruzione del tipo
  `cin >> x >> y >> z;`
- con uno stream contenente
  `.....□□a□-14.53□□728□□.....`
  ↑ ⟶
- se **`x`**, **`y`** e **`z`** sono rispettivamente di tipo **`char`**, **`double`** e **`int`**
- al termine dell'esecuzione **`x`**, **`y`** e **`z`** avranno rispettivamente i valori **`'a'`**, **`-14.53`** e **`728`**

C++: output

- Lo stream di uscita standard (**`cout`**) è quello su cui il programma scrive i dati ed è tipicamente associato allo schermo
- Per scrivere dati si usa l'istruzione di scrittura
  `stream << espressione;`
- L'istruzione di scrittura comporta
  - il calcolo dell'espressione
  - la conversione in una sequenza di caratteri
  - il trasferimento della sequenza nello stream
- L'istruzione di scrittura ha una forma più generale che consente scritture multiple
- Esempio:

```
cout << x << y << endl;
```

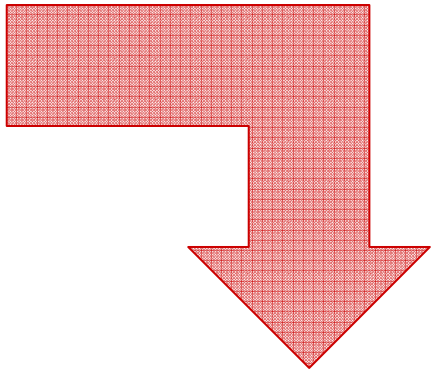
  equivale a

```
cout << x;
cout << y;
cout << endl;
```

- Nota: la costante predefinita **`endl`** corrisponde ad un **`'\n'`** per cui viene iniziata una nuova linea con il cursore nella prima colonna

C++: I/O example

```
#include <iostream>

using namespace std;
void main() {
  int age,year;
  cout << "Quanti anni hai? ";
  cin >> age;
  cout << "In che anno siamo? ";
  cin >> year;
  cout << "Sei nato nel "
       << year-age << endl;
}
```



```
Quanti anni hai? 25
In che anno siamo? 2000
Sei nato nel 1975
```

C++: uso di files

- Un'apposita libreria del C++ consente di utilizzare stream che vengono associati a file del sistema operativo
- È utilizzabile con la direttiva
  `#include <fstream>`
- stream di questo tipo vengono dichiarati con
  `fstream` *nomeidentificatore*`;`
- Ad esempio:
  `fstream ingresso, uscita;`
- Uno stream associato ad un file può essere aperto mediante la funzione **`open()`** secondo le modalità, identificate dalle costanti tra ( )
  - lettura (**`ios::in`**)
  - scrittura (**`ios::out`**)
  - append (scrittura alla file del file) (**`ios::app`**)
- Il nome del file viene specificato come una sequenza di caratteri (es. **`"file1.in"`**)
- Esempio:

```
fstream ingr, usc;
ingr.open("file1.in", ios::in);
usc.open("file2.out", ios::out);
```

C++: uso di files

- Uno stream, quando è stato utilizzato, può essere chiuso mediante la funzione **`close()`**
- Esempio:

```
ingr.close();
usc.close();
```

- Alla fine del programma tutti gli stream aperti vengono automaticamente chiusi
- È buona norma chiudere uno stream se non più utilizzato.
- Una volta chiuso, uno stream può essere riaperto in qualunque modalità e associato a qualunque file

C++: funzioni librerie per gli stream

- **`cin.get(x);`** legge tutti i caratteri, compresi anche quelli di spaziatura; se **`x`** non è di tipo **`char`** è equivalente a **`cin>>x;`**
- **`cin.eof();`** se lo stream **`cin`** ha raggiunto la sua fine (End Of File) viene ritornato un valore diverso da 0
- **`cin.fail();`** segnala uno stato di errore, assumendo un valore diverso da 0; accade quando si è tentato di leggere una sequenza di caratteri non consistente con il tipo di variabile dell'istruzione di lettura (es. si cerca di leggere un **`int`** e in input ci sono delle lettere)
- **`cin.clear();`** effettua il ripristino dello stato normale dallo stato di errore

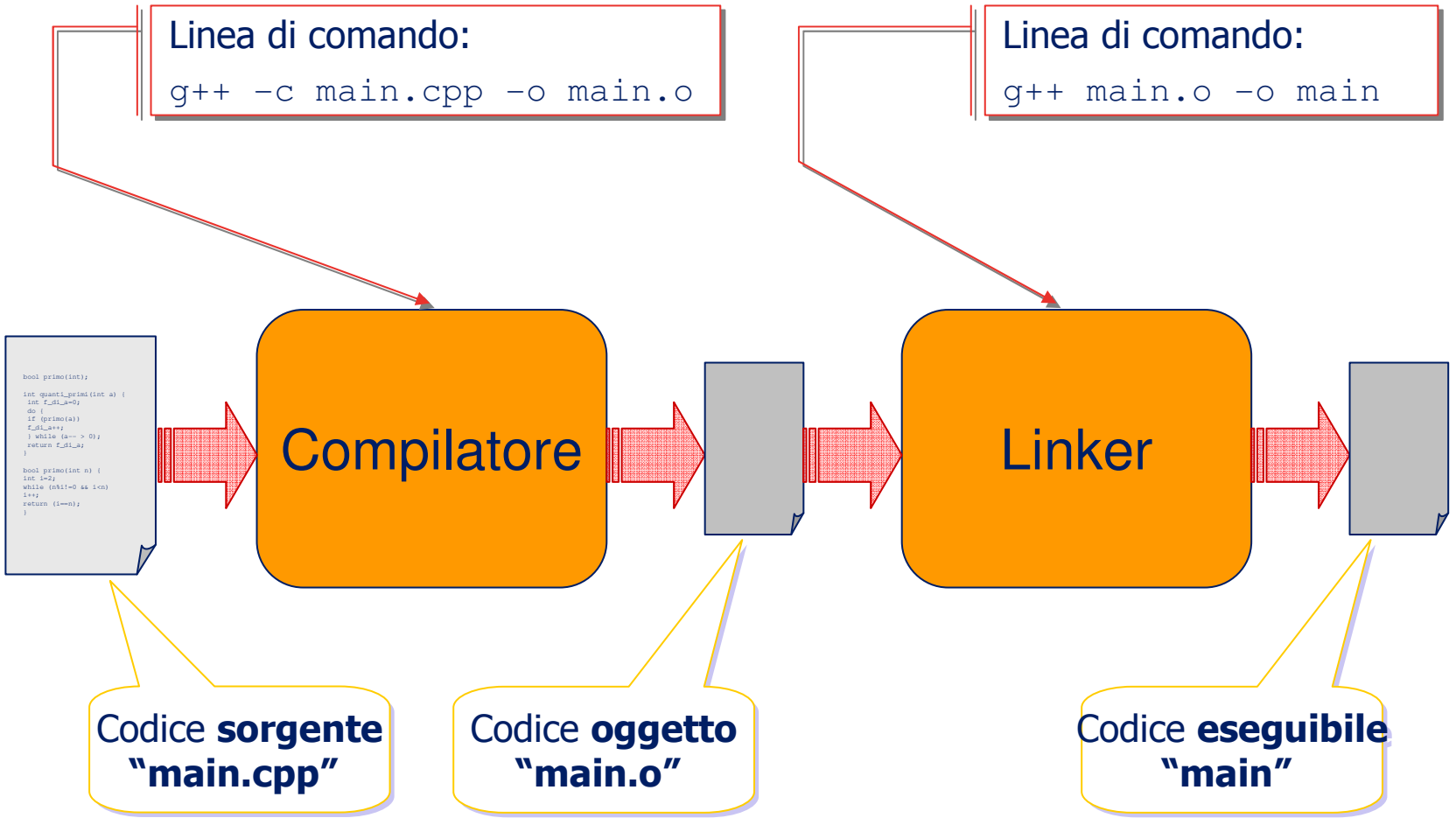
C++: struttura di un programma

- Un programma C++ consiste in un insieme di funzioni, eventualmente suddivise in più files.
- La funzione che costituisce il programma principale si deve chiamare **main.**
- Contiene una lista di istruzioni di ogni tipo, semplici o strutturate.
- Esempio:

```
int main() {
    int x = 2, y = 6, z;
    x = x*y;
}
```

Questo è il contenuto del programma principale

C++: compilazione di un programma



C++: Suddivisione del codice

```
#include <iostream>

using namespace std;

#include “array.h”

int main () {
   int val[10];
   riempi(val,10);
   stampa (val,10);
   cout << somma (val,10) << endl;
}
```

main.cpp

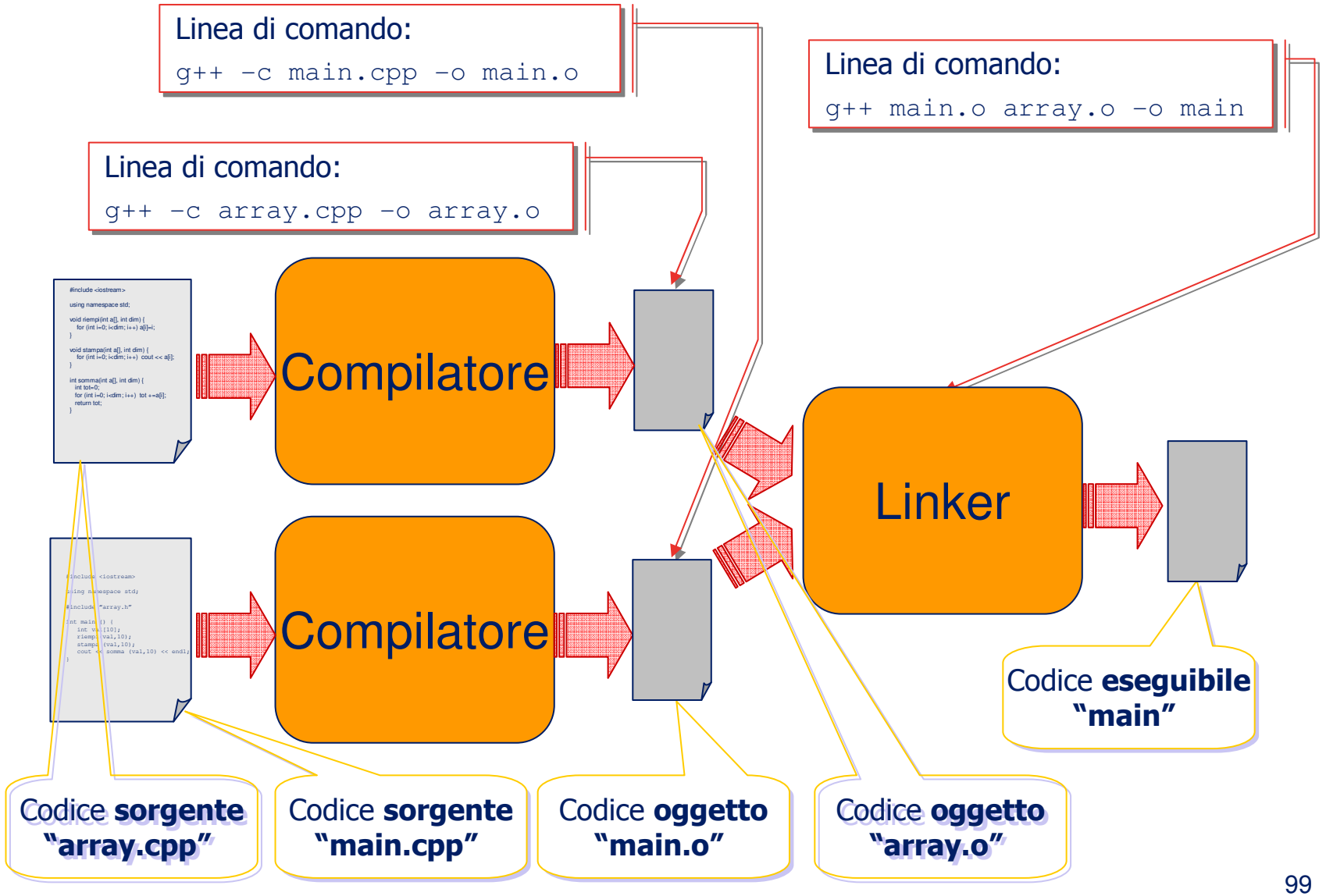
```
void riempi(int a[], int dim);
void stampa(int a[], int dim);
int somma(int a[], int dim);
```

array.h

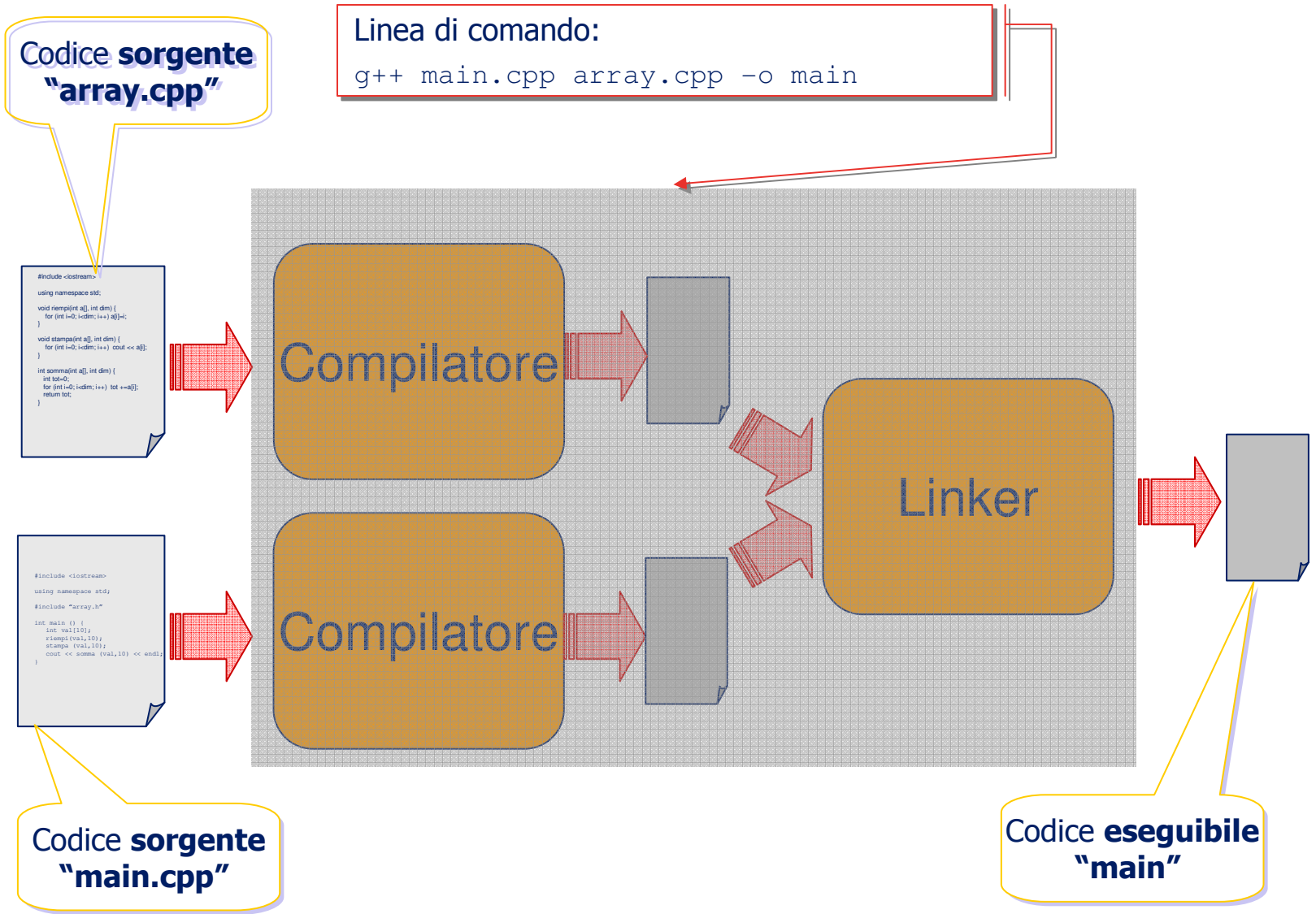
```
#include <iostream>

using namespace std;

void riempi(int a[], int dim) {
    for (int i=0; i<dim; i++) a[i]=i;
}

void stampa(int a[], int dim) {
    for (int i=0; i<dim; i++)  cout << a[i];
}

int somma(int a[], int dim) {
   int tot=0;
   for (int i=0; i<dim; i++)  tot +=a[i];
   return tot;
}
```

array.cpp

C++: compilazione di un programma



C++: compilazione di un programma



Esercizi

- Scrivere un programma C++ che, dati tre numeri in input dall'utente, calcoli il massimo, il minimo e il valore medio dei tre numeri;
- Scrivere un programma C++ che risolva lo stesso problema di prima, ma ricevendo in input dall'utente una sequenza arbitrariamente lunga di numeri positivi (l'immissione dei dati termini quando l'utente immette il valore "-1");
- Scrivere un programma C++ che, dati tre numeri in input dall'utente, li ristampi in ordine dal più piccolo al più grande;
- Scrivere in C++ un programma che calcoli le soluzioni dell'equazione "ax+b=0" (dati in ingresso dall'utente i valori dei coefficienti a e b), tenendo conto dei casi speciali "infinite soluzioni" e "nessuna soluzione"; modificare poi l'algorimo per l'equazione di secondo grado in modo che tenga conto degli analoghi casi limite;

Esercizi

- Scrivere in C++ un programma che, dato in ingresso un intero, stampi sullo schermo un “albero di natale”, alto un numero di righe pari all’intero immesso.

  As esempio, se l’utente immette il valore “6”, la risposta sarà:

```
     *
    ***
   *****
  *******
 *********
***********
```

- Scrivere un programma C++ che, dato in ingresso un intero, produca in uscita la fattorizzazione di quell’intero.

  Ad esempio, se l’utente immette “121680”, la risposta sarà qualcosa del tipo:

  121680 = 2^4 * 3^2 * 5 * 13^2

Esercizi

- Scrivere un programma che, presa in considerazione una stringa rappresentata come array di caratteri, cambi ogni lettera minuscola nella corrispondente maiuscola e viceversa, lasciando inalterati spazi e segni di interpunzione.

  Ad esempio da: Ciao a TUTTI!
  si otterrà: cIAO A tutti!

  – *Suggerimento:*

    Si sfrutti il fatto che la distanza sulla tabella ASCII tra una lettera maiuscola e la corrispondente minuscola è fissa e nota (e pari a 32).

- Scrivere una funzione ricorsiva e la sua versione non ricorsiva che calcola la somma dei primi n numeri interi;