Programmazione (00819)

Alberto Zuccari

A.A 2021-2022

Indice

Chapter 1

Ide, dichiarazioni, tipi

1.1 Identificatori

Gli **identificatori** (o variabili) sono **nomi simbolici** creati dal programmatore ed associati ad un *valore*.

Gli identificatori possono essere **sequenze** di lettere, cifre e il simbolo "`_`" *non* possono iniziare con una cifra.

Esempi:

```
Gino_66      X27      un_identificatore
```

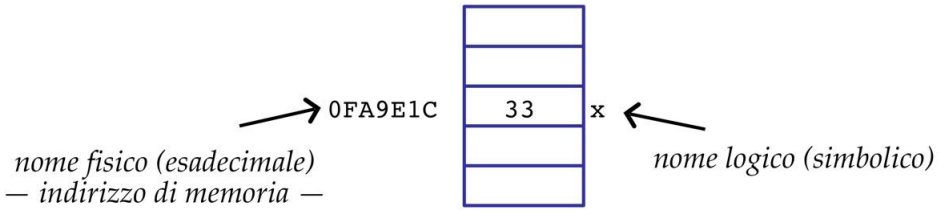
Attenzione!

1. `C++` è sensibile al tipo dei caratteri (se minuscolo o maiuscolo)

   `un_ide` `UN_IDE` `Un_IdE` sono differenti

2. non si possono usare le parole chiavi del `C++` (`int, float, double,` ...) nè gli identificatori standard (`main, while,` ...)

Fisicamente un identificatore è il nome associato ad una cella di memoria utilizzata per contenere valori



memoria

(sequenza di celle i cui indirizzi sono numeri esadecimali)

Il programmatore non può conoscere il nome fisico della cella di memoria perché viene deciso solamente durante l'esecuzione del programma.

1.2 Dichiarazioni

Le dichiarazioni sono una parte di programma che comunica al programma 2 cose:

1. gli identificatori utilizzati
2. il **tipo** dei valori da memorizzare in ogni identificatore

Facciamo un esempio: `double kms, miles;`

- gli identificatori sono `kms` e `miles`
- il tipo è `double` (**numeri reali**)

Sintassi:

```
int x                    // interi
double y                 // reali
char z                   // caratteri
```

Le dichiarazioni servono ad **allocare la memoria sufficiente** a contenere i valori utilizzati dal programma, diciamo quindi che *le dichiarazioni non hanno alcun effetto "visibile"*

1.3 Tipi di dato

I valori manipolati dai programmi sono suddivisi in insiemi disgiunti, detti **tipi di dato**

I tipi di dato servono a **ottimizzare l'uso della memoria**

Tipo di dato	Significato	Dimensione (in Byte)
`int`	numero intero	2 o 4
`float`	numero a virgola mobile	4
`double`	numero reale	8
`char`	carattere alfanumerico	1
`wchar_t`	carattere largo	2
`bool`	valore booleano	1
`void`	vuoto	0

1.3.1 int

valori

Sono il sottoinsieme che è possibile memorizzare in 4 byte

valori compresi tra `±` `2.147.483.647`

esempi: `-16` `0` `3257` `21`

Operazioni

- **memorizzare un intero** in una variabile di tipo `int`:
  `x = 21`
- effettuare **operazioni aritmetiche** (somma, differenza, moltiplicazione, divisione, resto, ...) tra due interi:
  `5 + 4` `4/2` `5%2` `4 * 7`
- **confrontare due interi**:
  `5 > 7` `5 == 4` `5 != 4` `5 >= 4`

1.3.2 double

Valori

Sono un'astrazione dei reali: alcuni reali sono troppo grandi o troppo piccolo, mentre altri non possono essere rappresentati in modo preciso

esempi	**notazione decimale**	**notazione scientifica**
	`3.1414`	`31.415e-1`
	`0.000016`	`0.16E-4`
	`120.0`	`12e1`

Valori che non sono `double`:

`150`	non c'è il punto
`3,45`	la virgola non è consentita
`2e.3`	`.3` non è un esponente valido
`13e`	manca l'esponente

Operazioni

- **memorizzare** un reale in una variabile di tipo `double`
- **operazioni aritmetiche** (somma, differenza, moltiplicazione, divisione)
- **confronto**
- **operazioni di libreria**

1.3.3 cast

L'operazione di cast consiste nella conversione di un intero in un reale o viceversa

```
(int)4.6    →   4
(double)4   →   4.0
```

1.3.4 char

Valori

Sono i singoli caratteri (lettere, cifre, simboli speciali) ogni valore di tipo char è racchiuso da apostrofi

esempi: `'A'` `'b'` `'7'` `';'`

Operazioni

- **memorizzare** un carattere in una variabile di tipo char
- **confrontare** due caratteri

esempi: `'A' > 'b'` `'a' >= '@'` `'a' == '@'`

1.3.5 bool

valori

`true` e `false`

operazioni

- `& &` and logico
- `||` or logico
- `!` not logico

esempi: `true||false` `('A' > 'b') && (!(3>4))`

1.4 Assegnamenti

L'istruzione di assegnamento memorizza un valore o il risultato di un calcolo in una variabile

sintassi:

variabile `=` espressione`;`

Esempio:

```
x = a * b;
```

- calcola il valore dell'espressione `a * b`
- se il calcolo dell'espressione **non produce errori** ⇒ il valore viene **assegnato** alla variabile `x`

1.5 Espressioni

Una espressione è una sequenza di operazioni che restituiscono un valore.

Una *espressione* può essere:

- una **variabile**
- una **costante**
- una **chiamata di funzione**
- una **combinazione** di variabili e costanti (e chiamate di funzioni) connesse da operatori

esempi: `(5-2)-4` `y+(x*5)` `7*funct(x, y) + sqrt(144)`

1.5.1 Valutazione delle espressioni

Come valutiamo l'ordine di lettura per una espressione?

L'ordine di valutazione delle espressioni è fissato da:

- **parentesi**
- **precedenza tra operatori**
  - operatori con la stessa precedenza vengono valutati da sinistra verso destra

1.5.2 Tipo di una espressione

Qual è il tipo di una espressione?

risposta: è determinato dalle **operazioni** e dal **tipo degli operandi**

esempio:

```
x + y
```

- se entrambi `x` e `y` sono di tipo `int` all'ora l'espressione ha tipo `int`
- se `x` o `y` hanno tipo `double` (gli operatori aritmetici `+`, `-`, `*`, `/` sono overloaded)

1.6 type safety

nei linguaggi di programmazione i tipi vengono utilizzati per rilevare errori del programmatore

La `type safety` è un'implementazione del compilatore e dice che **ogni entità deve essere usata in accordo con il suo tipo**

- una variabile può essere usata solo **dopo** che è stata dichiarata
- solamente le operazoni definite per il tipo dichiarato per la variabile **possono essere applicate** ad essa
- ogni operazione (**totale**) applicata correttamente **ritorna un valore valido**

1.7 Esercizi

1. dato questo frammento di codice
   ```
   char x, y;
   cin>> x >> y;
   ...
   cout<< x << y;
   ```
   scrivere una sequenza di comandi che scambia il valore di due identificatori (quando gli identificatori son `int` e `double`, si può fare senza un terzo identificatore...)
2. scrivere un programma che prende in input tre reali e li stampa in modo invertito
3. scrivere un programma che calcola l'area di un cerchio dato il raggio

1.8 Costanti

Le costanti si dichiarano come le variabili ma aggiungendo il prefisso `const`

```
const double pi = 3.14, e = 2.71:
```

oppire si dichiarano con

```
#define g 9.81
```

Inizializzazione Non è possibile cambiare il valore di una costante (il compilatore da errore)
Per questo motivo le costanti devono essere inizializzate al momento della dichiarazione

1.9 C++ e la Matematica

Stiliamo una lista di regole da seguire quando applichiamo le conoscenze matematiche alla programmazione in `C++`:

- l'operatore `=` *non è l'uguaglianza* ma un **assegnamento**
- il segno di prodotto non può essere sottointeso
  `y = 2x` è errato
- per delimitare gli argomenti di una frazione si usano le parentesi
  `(3 * 5) / 2`
- in `C++` posso avere due operatori consecutivi, se il secondo è unario
  `4 * -2`

1.9.1 Funzioni di libreria

Il `C++` p dotato di librerie per calcolare le funzioni matematiche di uso più comune
Se includiamo `<cmath>` abbiamo:

- `double abs(double)`
- `double sqrt(double)`
- `double pow(double, double)`
- `double cos(double)`
- `double sin(double)`
- `...`

1.10 Numeri pseudo-casuali

Includendo `<cstdlib>` abbiamo:

- `int rand();`
- `int srand(int);`
- `RAND_MAX`

Scriviamo ora un programma che genera un numero random tra `0` e `89`

```
#include <iostream>
#include <cstdlib>
#include <ctime>

using namespace std;

int main(){     // srand() inizializza la funzione di generazione di numeri p-c
    srand( time(0) );       // time(0) restituisce il tempo
    cout<< rand() % 90;     // rand() genera i numeri pseudo-casuali
    return 0;
}
```

1.11 Esercizi

1. scrivere un programma che prende in input un prezzo in euro (inclusi centesimi) e stampa quante e quali monete sono necessarie per pagarle (con un numero minimo di monete)
   ```
   esempio:      input: 15.74
                 output: monete da 2 euro: 7
                         monete da 1 euro: 1
                         monete da 50 cent: 1
                         monete da 20 cent: 1
                         monete da 2 cent: 2
   ```
2. scrivere un programma che prende in input la lunghezza di due cateti di un triangolo rettangolo e stampa la lunghezza dell'ipotenusa

1.12 Input/Output

I costrutti di `input/output` sono fodamentali per interagire con il programma.

In `C++` un modo semplice e potente per fare `input/output` è dato dagli stream `cin` e `cout` (uno **stream** è una sequenza di caratteri)

1.12.1 L'operatore di output

L'operatore `<<` invia i valori al canale di output specificato (useremo solamente `cout`)

sintassi:

```
cout << exp;    // calcola exp e scrive il valore di exp in cout
```

L'operatore `<<` è in grado di stampare dati di (quasi) tutti i tipi base

```
int x = 5;
cout<<"x = ";
cout<<x;
```

Possiamo comporre l'operatore `<<` per stampare più valori con un'unica istruzione

```
cout<<"x = " << x;
```

L'operatore `<<` associa a sinistra e restituisce lo stream stesso.
Il livello di precedenza è **inferiore** a quello degli operatori aritmetici quindi è permesso l'utilizzo di espressioni aritmetiche senza parentesi:

```
cout<<"x + y = " <<x+y;
```

1.12.2 sequenze di escape

Le sequenze di escape vengono utilizzate per inserire caratteri speciali.

Sono composte da backslash (\) seguito da un codice speciale

- `\n` nuova linea
- `\t` tab
- `\\` backslash

Si utilizza anche il manipolatore `enld` (per esempio sul programma base `Hello World` su `Eclipse`).

`endl` inserisce `\n` nello stream e stampa a video eventuali caratteri rimasti nello stream

1.12.3 L'operatore di input

L'operatore `>>` riceve valori da un canale di input (useremo solamente `cin`) salvandoli nel secondo argomento

sintassi:

```
cin >> exp;   // legge da cin il valore in exp
```

Possiamo comporre l'operatore `>>` per leggere più valori con un'unica istruzione

```
int x, y;
cin>> x >> y;
```

Associatività e Precedenza di `>>` sono **uguali** a `<<` e anche lui restituisce lo stream.
L'operatore `>>` ha un comportamento diverso a seconda del tipo di dato
esempio:

```
char x;
int y;
cin >> x >> y;
```

- poiché `x` è una variabile `char`, `cin` legge un carattere
- poiché `y` è una variabile `int`, `cin` legge finché non trova caratteri numerici validi
- se un identificatore è una variabile a virgola mobile legge finché non trova caratteri validi per valori a virgola mobile.
- eventuali **spazi o a capo iniziali** non vengono considerati

---

1.12.4 I/O: Esercizi

1. scrivere un programma che chieda in input la vostra età e il vostro sesso (come carattere M/F) e li stampi a video
2. riscrivere il programma precedente usando una sola volta `cin` e una sola volta `cout`
3. scrivere un programma che presi in input la base e l'altezza di un triangolo ne stampi l'area
4. scrivere un programma che prende in input 3 numeri interi e produce in output una tabella con la differenza dei numeri a due a due.
   esempio: se i numeri sono `2, 7, 3` deve stampare:

```
    2   7   3
2   0  -5  -1
7   5   0   4
3   1  -4   0
```

Chapter 2

comandi condizionali e iterativi

2.1 Condizionali

2.1.1 If-Then-Else

Il costrutto `if-then-else` permette di effettuare una scelta tra diversi comandi alternativi da eseguire.

La scelta viene fatta calcolando il valore di una espressione booleana detta *condizione*

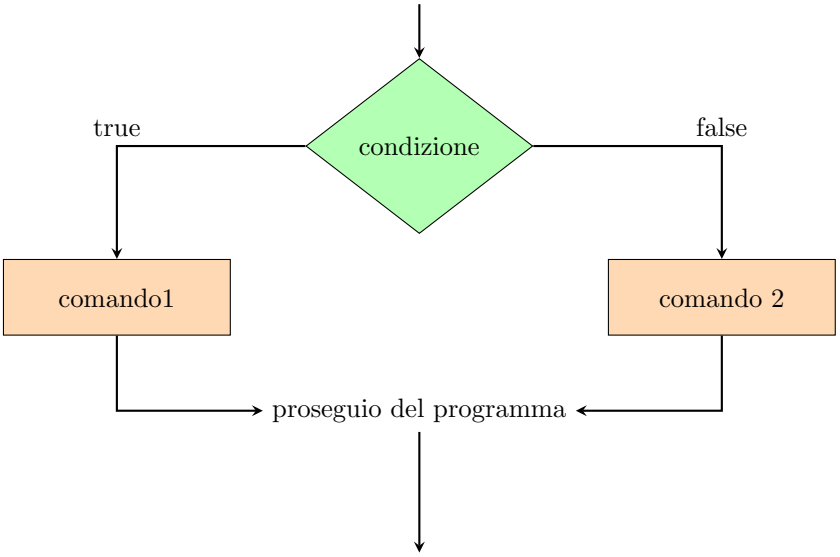
sintassi:

```
if (condizione){     // ramo if
    comando_1;       // ramo then
}else{
    comando_2;       // ramo else
}
```

esempio:

```
if ((x >= '0') && (x <= '9')){
    cout<<"carattere numerico";
}else{
    cout<<"simbolo";
}
```

semantica:



1. valuta la condizione
2. se il risultato è `true` esegue l'istruzione che segue la condizione (*ramo then*)
3. se il risultato è `false` esegue l'istruzione che segue la parola `else` (*ramo else*)

2.1.2 Condizioni

Comandi relazionali:

`<`	(minore di)	`>`	(maggiore di)
`<=`	(minore o uguale a)	`>=`	(maggiore o uguale a)

operatori di uguaglianza:

`==` (uguale a) `!=` (diverso da)

operatori logici:

`&&` (and) `||` (or) `!` (not)

Problemi

- la seguente espressione esprime la condizione "`x` è compreso fra `min` e `max`"?

```
min <= x <= max
```

- la seguente espressione esprime la condizione "`x` e `y` sono maggiori di `z`"?

```
x && y > z
```

- la seguente espressione esprime la condizione "`x` è uguale a `1.0` oppure a `3.0`"?

```
x == 1.0 || 3.0
```

Risposta:

No.

Per rendere le espressioni che scriviamo normalmente in matematica dobbiamo utilizzare al massimo le parentesi "`( )`".

2.1.3 If-Then

Il costrutto `if-then` permette di effettuare una scelta tra **un comando** da eseguire e non far nulla.
La scelta viene calcolata secondo la condizione booleana (nella stessa maniera di `if-then-else`).

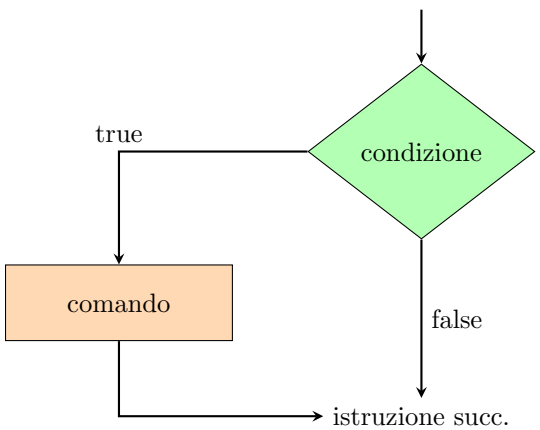
sintassi

```
if (condizione) comando;
```

esempio:

```
if (x != 0) cout<< 25/x;
```

semantica:



1. valuta la condizione tra parentesi
2. se il risultato è `true` esegue L'istruzione che segue la condizione (*ramo then*)
3. se il risultato è `false` va all'istruzione successiva

2.1.4 Sequenze di If

C'è la possibilità di "annidate gli `if`" ovvero scrivere a *cascata* una serie di condizioni che possono avvenire a seconda dei casi.

Per scriverlo è necessario porre dopo `else` un altro `if` in modo tale da far continuare il controllo su di un'altra condizione che da per falsa la precedente ma non chiude i casi di analisi.

formato standard

```
if (condizione_0){
    comando_0;
}else if (condizione_1){
    comando_1;
}
...
}else if (condizione_n){
    comando_n;
}else{
    comando_e;
}
```

esempio:

```
if (x > 0){
    num_pos = num_pos + 1;
}else if (x > 0){
    num_neg = num_neg + 1:
}else{
    num_zero = num_zero + 1;
}
```

2.1.5 If: Trappole

Tipici errori di programmazione (non di compilazione) legati all'`if`:

`if (x = 0) ...`

`x = 0` **non è una condizione booleana** ma un'associazione

Errori nelle condizioni come quelli visti in precedenza:

`x <= y <= z`

Le parentesi!!!

2.1.6 Esercizi:

1. scrivere un "firewall" che prende in input 5 caratteri e stampa solamente le lettere minuscole
2. scrivere un programma che prende 3 interi e stampa `1` se uno dei tre è divisore degli altri due, `0` altrimenti (fare due versioni: una con l'`if` e una senza)
3. scrivere un programma che simula una calcolatrice tascabile con le operazioni "+", "-", "*", "/" e "%"; cioè prende un intero, uno dei simboli precedenti, e un altro intero e calcola il risultato dell'operazione relativa
4. scrivere un programma che prende 3 interi e li stampa in maniera ordinata
5. scrivere un programma che prende 4 interi e stampa l'intero tra i 4 più vicino al valore medio

2.2 Iterativi

Vogliamo ora, per **esempio** stampare `10` volte "`ciao`"

```
int main(){
    cout<<"ciao" <<endl;
    ...
    ...
    ...
    cout<<"ciao" <<endl;
    return(0);
}
```

E volessi stampare `100` volte?

È impensabile scrivere così tante volte la stessa cosa!

2.2.1 while

Entra quindi in gioco il costrutto `while` che ci consente di ripetere un gruppo di comandi, detto **corpo del ciclo**

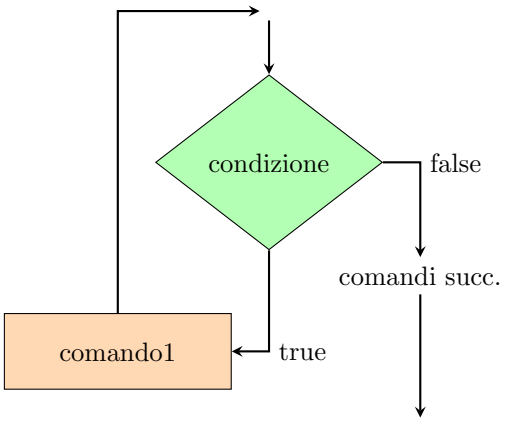
"*finché*" una determinata condizione, detta **guardia del ciclo** resta valida.

sintassi:

```
while (condizione){ // guardia del ciclo
    operazione_0;
    ...             // corpo del ciclo
    operazione_n;
}
```

- *condizione* è la struttura decisionale per controllare il numero di iterazioni
- *operazione* è il gruppo di istruzioni che vengono ripetute

semantica



1. valuta la guardia del ciclo
2. se il risultato è `true` esegue il corpo del ciclo e ritorna al punto 1.
3. se il risultato è `false` il comando `whil` termina e si passa a quello successivo.

Osservazioni:

- se la *condizione* è `false` la prima volta che viene testata l'*operazione* non viene eseguita
- se la *condizione* è **sempre** `true` l'iterazione non termina (ciclo **infinito**)
  Ogni iterazione deve quindi contenere una istruzione che invalida la guardia del `while`

esempio:

```
while (x != 12){
    cout<< x <<endl;
    x = x + 2;
}
```

meglio:

usare la condizione: `while (x < 12)`

2.2.2 for

Sintassi

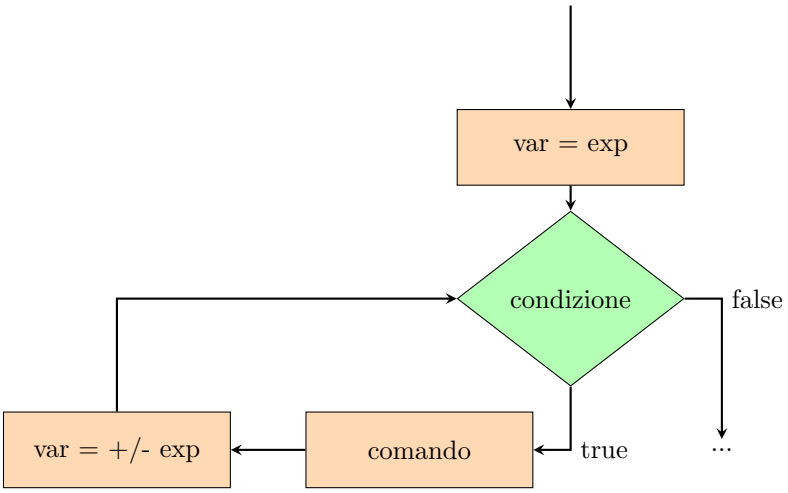
```
for (int i = 0; i < N; i++){
    operazione;
}
```

Esempio: stampare 10 volte "ciao"

```
for (int i = 0; i < 10; i++){
    cout<<"ciao" <<endl;
}
```

- `int i = 0;` è l'assegnamento di inizializzazione della *variabile di controllo*
- `i < N` è la struttura decisionale per controllare il numero di iterazioni, detta *guardia del ciclo for*
- `i++` è l'incremento della variabile di controllo (può essere modificato a seconda delle necessità del ciclo)

Semantica:



Il comando `for` permette di raggruppare *in un'unica posizione* tutte e tre le componenti tipiche di una iterazione:

1. inizializzazione della variabile di controllo del ciclo
2. test della condizione di ripetizione del ciclo
3. aggiornamento della variabile di controllo del ciclo

Osservazione:

Un comando `for` può essere riscritto in un comando `while` e **viceversa!**

```
for (int i=0; i<N; i++){

    comando;

}
```

```
int i = 0;
while (i<N){
    comando;
    i++
}
```

2.2.3 Differenza ed utilizzo

Il costrutto `for` va utilizzato:

- quando ***staticamente*** è noto il numero di iterazioni che il comando deve fare

Letteralmente *per TOT volte.*

Il costrutto `while` va utilizzato

- quando ***non sappiamo precisamente*** quante volte il ciclo va eseguito

Letteralmente *finché non avviene questo*

2.2.4 do-while

Sintassi

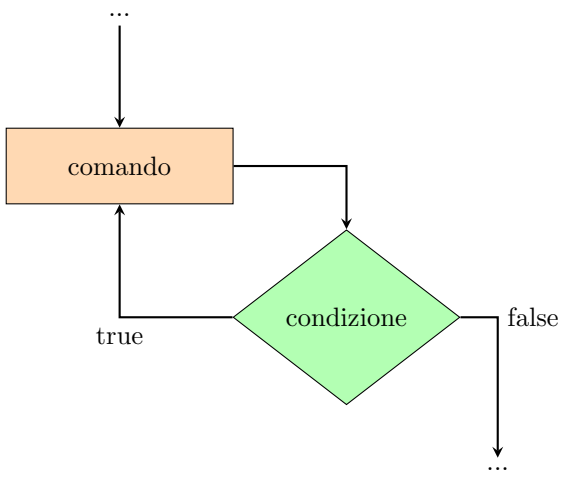
```
do{
    comando;
} while (condizione);
```

1. Viene eseguito il comando e **in seguito** la condizione viene testata
2. se è `true` il comando viene ripetuto
3. se è `false` si esce dal ciclo e l'esecuzione contina con il comando successivo al `while`

Osservazione:

Il corpo del `do-while` viene eseguito **almeno una volta** anche nel caso in cui la condizione è falsa

Semantica:



2.2.5 Esercizi

1. scrivere un programma che prende in input un numero intero `n` e stampa `n` asterischi.
2. scrivere un programma che chiede in input un numero: se è primo stampa "`primo`" altrimenti stampa "`non primo`"
3. scrivere un programma che interroga uno studente sulla divisibilità: prende in input un numero intero e chiede allo studente di inserire un divisore.

   Il programma termina solo quando lo studente inserisce un divisore corretto.
4. scrivere un programma che prende in input un intero e stampa la somma delle sue cifre

2.3 Cicili annidati

Spesso può essere utile usare un ciclo dentro l'altro

- ad ogni iterazione del ciclo *esterno* corrispondono **tutte le possibili iterazioni** del ciclo *interno*
- Bisogna fare attenzione a:
  - non mischiare le variabili di controllo dei `2` cicli
  - ripristinare i valori di inizializzazione del ciclo intrno ad ogni iterazione del ciclo esterno
- **attenzione** il programma diventa **poco leggibile**.

2.3.1 Esercizi

1. scrivere un programma che stampa un triangolo isoscele dell'altezza desiderata presa in input.
   Ad esempio se `h = 4` la stampa:
   ```
      *
     ***
    *****
   *******
   ```
2. scrivere un programma che chiede all'utente dei numeri interi e scrive se sono primi o non primi. Il programma termina quando l'utente inserisce il numero `0`

2.4 Controllo dei cicli

Ci sono **due** metodi per controllare le iterazioni dei cicli

1. controlli definiti da **Contatori**
    - è il tipo di controllo **più semplice** perché prima che il ciclo inizi si conoscono il numero di iterazioni
2. cicli controllati da **flag**
    - una variabile il cui cambiamento di valore indica che un particolare evento è avvenuto è detta **flag**

Consideriamo il seguente ciclo (Collatz 1937):

```
while (n != 1){
    if (n%2 == 0){
        n = n / 2;
    }else{
        n = 3 * n + 1;
    }
}
```

Siamo sicuri che termina?

Esiste una congettura secondo la quale, qualunque sia il numero iniziale questo programma **termina** sempre
Ma non è mai stata dimostrata

Poiché: il ciclo potrebbe non terminare, conviene imporre la terminazione tramite un `flag`

- utilizziamo una variabile `flag` inizializzata a un certo limite superiore
- ad ogni iterazione decrementiamo `flag`
- modifichiamo la guardia in

`((n != 1) && (flag > 0))`

Il ciclo di Collatz diventa:

```
int flag = upper_bound; // un limite di cicli da eseguire
while ((n != 1) && (flag > 0)){
    if (n%2 == 0){
        n = n / 2;
    }else{
        n = 3 * n + 1;
    }
    flag = flag - 1;
}
```

Nota: il **flag** molte volte è un **booleano**

Chapter 3

Funzioni

Le funzioni sono parte di codice che teniamo separato dal `main` e richiamiamo quando vogliamo.

La suddivisione in componenti elementari, ognuno dei quali è una porzione del tutto, rende la gestione più facile.

Si tratta della famosa tecnica *dividi et impera*

3.1 Portata di una dichiarazione (scope)

La portata di una dichiarazione è la parte di programma in cui una dichiarazione di un identificatore è valida

Tra i comandi c'è il blocco: `{` `}`

Che di solito si usa per *raggruppare i comandi*

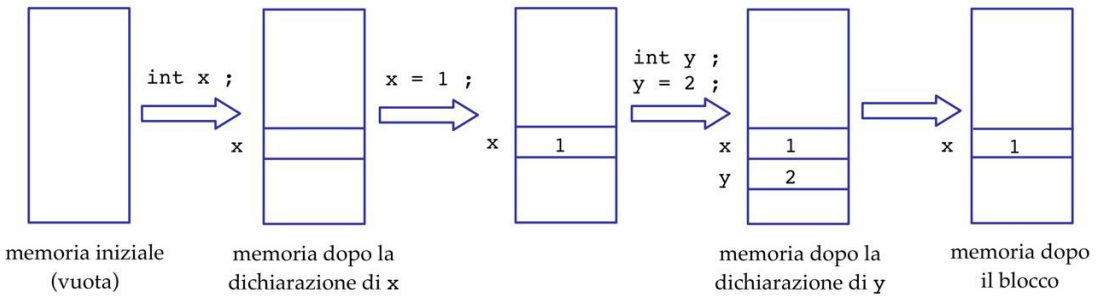
osservazione: in un blocco è possibile dichiarare nuovi identificatori, o richiamarne di vecchi!

Non è mai possibile dichiarare due volte lo stesso identificatore

3.1.1 Evoluzione della memoria

Evoluzione della memoria per il programma

```
1 int x;
2 x = 1;
3 {
4     int y;
5     y = 2;
6 }
7 cout<< x << y;
```



Quindi `cout<< x << y;` verrebbe eseguito senza che `y` sia dichiarata.

- la portata di una dichiarazione di un identificatore è il blocco in cui occorre la dichiarazione, *dal momento in cui occorre*, e tutti i blocchi interni a tale blocco
- se un identificatore dichiarato nel blocco `A` (ln `1-7`) è **ridefinito** nel blocco `B` (ln `3-6`) allora `B` e **ogni blocco interno a B** non fanno parte della portata della dichiarazione di `A`

3.2 Funzioni

Problema: introdurre un meccanismo che consente di riutilizzare codice già esistente

Soluzione: le funzioni (o sottoprogrammi)

Definizione: una funzione è un blocco (dichiarazioni + comandi) a cui è associato un nome

```
1 void radice_quadrata(){
2     int m, i;
3     cin>> m;
4     while (i*i <= m){
5         i = i + 1;
6     }
7     cout<<i-1;
8 }
```

osservazione: le funzioni possono avere degli argomenti e restituire un risultato

Benefici:

1. **evitare ripetizioni dello stesso codice**: se `radice_quadrata` devo calcolarlo in molti punti del programma, posso incapsulare il calcolo in una funzione e poi richiamarla
   (a) la lunghezza del codice diminuisce
   (b) il codice è più leggibile
   (c) si evitano errori
2. **il codice è decomposto in modi disgiunti**
   (a) facilita la comprensione del programma
   (b) facilita l'implementazione del codice
   (c) facilita le modifiche poiché si localizza il codice da modificare

3.2.1 Sintassi

Dobbiamo distinguere **due momenti:**

1. La **definizione** della funzione `function-definition`
2. L'**utilizzo** della funzione `function-call`

Definzione della funzione

```
function-definition ::= type identificatore (parametri formali){
                        declaration & statement
                        }
```

- La prima linea è detta **intestazione della funzione**
- La parte restante è detta **corpo della funzione**
- `type` indica il tipo di ritorno (`int`, `void`, ...)

Chiamate della funzione

```
function-call ::= identifier (parametri formali);
```

- le `function-call` **sono le funzioni che ritornano valori** e che devono essere usate all'interno di espressioni

Invocazioni void

`void-function-call ::= identifier (parametri formali);`

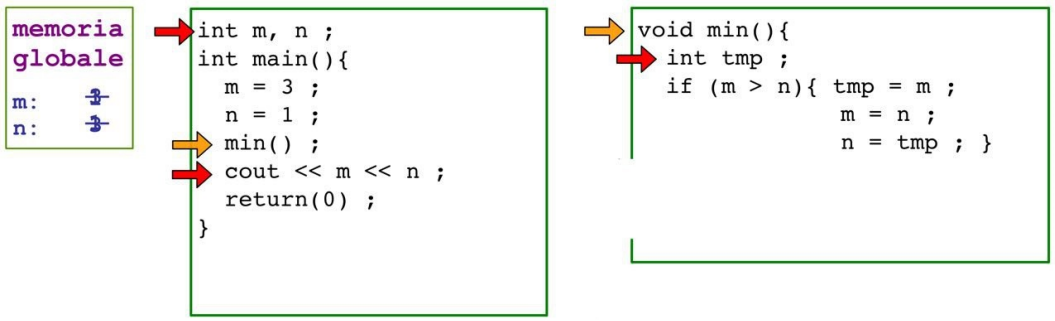
- le `void-function-call` **sono le funzioni che non ritornano niente** e che si usano come comandi.

3.3 Definizioni

Le funzioni sono definite allo stesso livello della funzione `main` (non è possibile *definire* una funzione all'interno di `main` o all'interno di un'altra funzione)

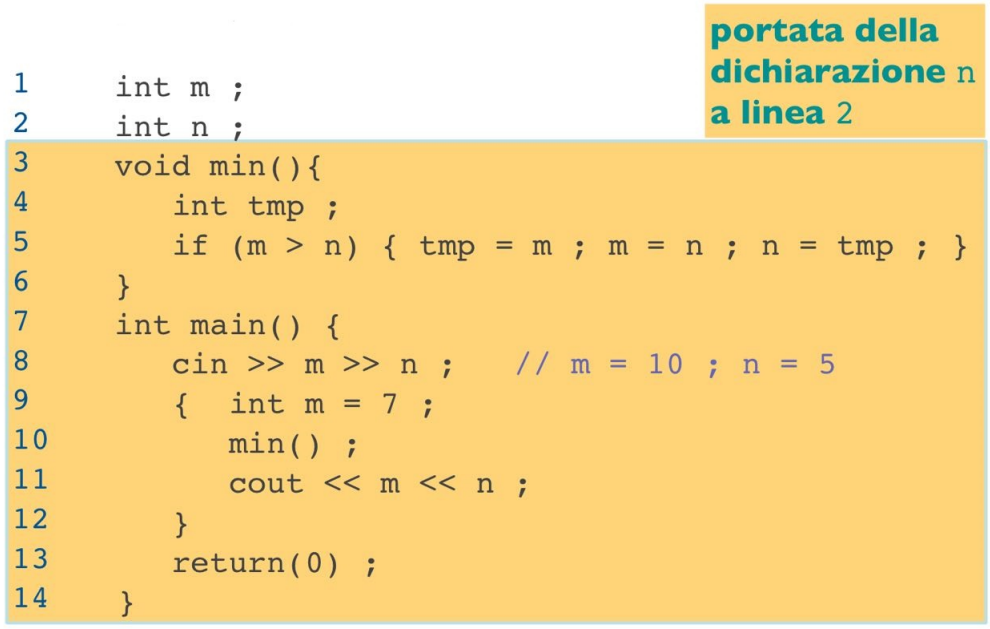
3.3.1 Semantica:

Ogni chiamata a funzione viene tradotta in **trasferimento del controllo** al codice della funzione



3.4 Portata di una dichiarazione

Vediamo adesso alcuni esempi delle portate di dichiarazioni di identificatori su di un programma



```
1     int m ;
2     int n ;
3     void min(){
4        int tmp ;
5        if (m > n) { tmp = m ; m = n ; n = tmp ; }
6     }
7     int main() {
8        cin >> m >> n ;    // m = 10 ; n = 5
9        {  int m = 7 ;
10          min() ;
11          cout << m << n ;
12       }
13       return(0) ;
14    }
```

portata della dichiarazione m **a linea** 9

portata della dichiarazione m **a linea** 1

```
1     int m ;
2     int n ;
3     void min(){
4        int tmp ;
5        if (m > n) { tmp = m ; m = n ; n = tmp ; }
6     }
7     int main() {
8        cin >> m >> n ;    // m = 10 ; n = 5
9        {  int m = 7 ;
10          min() ;
11          cout << m << n ;
12       }
13       return(0) ;
14    }
```

3.5 Funzioni con parametri

Lo svantaggio di una funzione come `min` è che vengono applicate solamente sempre alle stesse variabili (in questo caso variabili globali)

Vorremmo però far passare più parametri per andare a svolgere operazioni specifiche su identificatori definiti.

Occorre quindi, **durante la definizione di una funzione**, specificare la **lista dei parametri formali** che verranno presi in `input` dalla funzione.

```
1 void somma(int a, int b){
2     c = a + b;
3 }
4 void radice_quad(int n){
5     int i = 0;
6     while(i*i <= n) i = i + 1;
7     coutz<< i - 1;
8 }
```

I parametri formali sono identificatori a cui è associato un tipo, quando si effettua l'*invocazione* di una funzione occorre elencare gli **operandi** con cui saranno stanziati i parametri formali.
Gli operandi sono detti: **parametri attuali**

```
1 void stampa_int(int x){
2     cout<< x + 1;
3 }
4 int main(){
5     stampa_int(6);
6     return(0);
7 }
```

```
stampa 7
```

3.5.1 Vincoli sintattici

Esempio:

- la lunghezza della lista dei parametri attuali deve essere uguale alla lunghezza della lista dei parametri formali
- ogni parametro attuale corrisponde al parametro formale che occupa la stessa posizione nella lista dei parametri formali:

```
int funct(int a, int b, double c, float d){
    funct(6, 9, 3.14, 75.963)
```

  In questo esempio: `6` → `a` oppure `3.14` → `c`
- il tipo del parametro attuale deve essere uguale al tipo del parametro formale corrispondente

3.5.2 Ritorno di valori

Per far ritornare un valore al chiamante dobbiamo:

- specificare un tipo `non-void` nell'intestazione della funzione
  Sarà il tipo del valore ritornato
- inserire nel corpo l'istruzione `return`

Esempi:

```
1 int sum (int m, int n){
2     return(m + n);
3 }
4 int abstract(int m){
5     if (m < 0) return(-m);
6     else return(m);
7 }
```

L'espressione che è argomento di `return` viene calcolata ed il valore viene **ritornato** al chiamante

Quando si arriva ad un `return` 'esecuzione della funzione si **conclude** ed il controllo del programma ritorna al chiamante (ln `5-6` se la condizione `m < 0` si avvera viene ritornato `-m` e la funzione finisce)

3.5.3 Passaggio dei parametri

`C++` offre **tre** modalità di passaggio dei parametri:

1. per **valore**
2. per **riferimento**
3. per **costante**

Passaggio per valore

- i parametri attuali sono valutati
- il loro *valore* è memorizzato in **variabili locali alla funzione** che corrispondono ai parametri formali
- ogni modifica all'interno del corpo della funzione riguarderà le variabili locali

Esempio:

```
1  void scambia (int x, int y){
2      int tmp;
3      tmp = x;
4      x = y;
5      y = tmp;
6  }
```

```
7  int main (){
8      int a = 1, b = 2;
9      scambia (a, b);
10     cout<< a << b;
11     return 0;
12 }
```

`stampa 1 2` (e non 2 1)

Passaggio per riferimento

- i parametri formali devono essere dichiarati con "`&`" dopo il tipo
- i parametri attuali devono essere variabili (identificatori o similari)
- ogni modifica all'interno riguarderà i **parametri attuali**

Esempio:

```
1  void scambia (int& x, int& y){
2      int tmp;
3      tmp = x;
4      x = y;
5      y = tmp;
6  }
```

```
7  int main (){
8      int a = 1, b = 2;
9      scambia (a, b);
10     cout<< a << b;
11     return 0;
12 }
```

`stampa 2 1` (e non 1 2)

Errori nel passaggio per riferimento

```
1  void scambia (int& x, int& y){
2      int tmp;
3      tmp = x;
4      x = y;
5      y = tmp;
6  }
```

```
7  int main (){
8      int a = 1, b = 2;
9      scambia (a + b, b);
10     cout<< a << b;
11     return 0;
12 }
```

Alla linea 9 **NON si possono passare per riferimento delle espressioni**

Passaggio per costante

- I parametri formali devono essere dichiarati con "`const`" prima del tipo
- all'interno del corpo della funzione i parametri passati per costante **NON possono essere modificati**

Esempio:

```
int foo (const int x){
    int tmp;
    tmp = x * x;
    return(tmp);
}
```

```
int main (){
    int a = 2;
    cout<< foo(a + a);
    return(0);
}
```

3.6 Regole di programmazione

Evitare sempre funzioni che **modificano variabili globali** perché:

- complicano la comprensione del programma
- rendono ambiguo il significato

Evitare sempre di **interrompere una iterazione** (`for`, `while`) con un `return`
Aumenta la leggibilità del programma.

3.7 Inclusione di Librerie

Vogliamo utilizzare identificatori presenti in altri file
Per farlo scriviamo all'inizio del nostro programma:

```
// file library.h
void min (int& x, int& y) ;
```

```
// file library.cpp
#include "library.h"
void min(int& a, int& b){
      int tmp ;
      if (a>b) { tmp = a ; a = b ; b = tmp ; } ;
}
```

```
// file working.cpp
#include <iostream>
#include "library.h"
using namespace std;

int main (){
      int a=2, b=1;
      min(a,b) ;
      cout << a << b ;
      return(0) ;
}
```

Includere le librerie può però causare errori:
e includiamo anche `#include "librarybis.h"` nel main:

```
// file librarybis.h
void min (int& x, int& y) ;
```

```
// file librarybis.cpp
#include "librarybis.h"
void min(int& a, int& b){
      int tmp ;
      if (a>b) { tmp = 2*a ; a = 2*b ; b = tmp ; } ;
}
```

errore perché `min` è definito in due librerie. (presenta dichiarazioni multiple)

3.7.1 Namespace

I **namespace** consentono di risolvere questi problemi

```
// file library.h
namespace one {
   void min (int& x, int& y) ;
}
```

```
// file librarybis.h
namespace two {
   void min (int& x, int& y) ;
}
```

```
// file library.cpp
#include "library.h"
namespace one {
   void min(int& a, int& b){
      int tmp ;
      if (a>b) { tmp = a ; a = b ; b = tmp ; } ;
   }
}
```

```
// file librarybis.cpp
#include "librarybis.h"
namespace two {
   void min(int& a, int& b){
      int tmp ;
      if (a>b) { tmp = 2*a ; a = 2*b ; b = tmp ; } ;
   }
}
```

```
// file working.cpp
#include <iostream>
#include "library.h"
#include "librarybis.h"
using namespace std;

int main (){
      int a=2, b=1;
      one::min(a,b) ;
      cout << a << b ;
      return(0) ;
}
```

- un **namespace** è uno scope "etichettato" da un identificatore
- la sintassi "**::**" è usata per **specificare** quale namespace si sta usando, tra tutti quelli possibili

3.7.2 Using

- per evitare di scrivere `one::min(a, b)` si può usare la dichiarazione `using`

```
// file working.cpp
#include <iostream>
#include "library.h"
#include "librarybis.h"
using namespace std;

using one::min;

int main (){
      int a=2, b=1;
      min(a,b) ;
      two::min(a,b) ;
      cout << a << b ;
      return(0) ;
}
```

per accedere al namespace `two`

- oppure si può utilizzare la direttiva `namespace`

```
// file working.cpp
#include <iostream>
#include "library.h"
#include "librarybis.h"
using namespace std;

using namespace one ;

int main (){
      int a=2, b=1;
      min(a,b) ;
      two::min(a,b) ;
      cout << a << b ;
      return(0) ;
}
```

3.8 Esercizi

1. scrivere una libreria che contiene le funzioni
   - `bin2dec` prende un numero binario e ritorna il suo valore in base 10
   - `dec2bin` prende un numero decimale e ritorna il suo valore in base 2
   - `bin_sum` prende due numeri in binario e ritorna la loro somma in binario
   - `bin_prod` prende due numeri binari e ritorna il loro prodotto binario
2. scrivere una funzione "`ln`" che prende in input un double `1+x, -1 < x <= 1` e ritorna il logaritmo naturale definito dalla formula:

$$\ln(1+x) = \sum_{n=1}^{+\infty} \frac{(-1)^{n+1}}{n} x^n = x - \frac{x^2}{2} + \frac{x^3}{3} - \frac{x^4}{4} + \cdots$$

   (Iterarla fino a 50)
3. scrivere una funzione che risolve le equazioni di 2° grado $ax^2 + bx + c = 0$ cioè prende in input `a, b, c` e ritorna:

$$\frac{-b \pm \sqrt{b^2 - 4ac}}{2a}$$

Chapter 4

Array

Vogliamo ora "*raggruppare*" quantità di valori dello stesso tipo in un unico insieme, qui entrano in gioco **gli array** (o vettori)

Facciamo un esempio:

Scrivere una funzione che legge 20 interi e stampa il valore più vicino alla loro media:
Senza array:

- definire 20 variabili di tipo `int` che memorizzano gli interi presi in input
- calcolare la media
- stampare la variabile che è più vicina alla media

```
12 int average20(){
13     int var0, ..., var19;
14     int r, dist;
15     double m;
16     cin>> var0; ...; cin>> var19;
17     m = (var0 + ... + var19)/20;
18     r = var0; dist = abs(m - var0);
19     if (abs(m - var1) < dist) { dist = abs(m - var1); r = var1; }
20         ...
21     if (abs(m - var19) < dist) { dist = abs(m - var19); r = var19; }
22     return(r);
23 }
```

Osservazione: sulle 20 variabili ripetiamo sempre le stesse operazioni.

Riformuliamo il problema: Scrivere una funzione che legge 20.000 interi e stampa il valore più vicino alla loro media

4.1 Array

Gli Array sono collezzioni di dati dello stesso tipo a cui viene associato un unico nome simbolico

- i dati appartenenti all'array sono detti **elementi** dell'arrya
- gli elementi dell'array vengono memorizzati in celle di memoria contigue

4.2 Dichiarazioni

4.2.1 Sintassi

```
type id[int_const];
```

- il `type` è il **tipo base**
- `int_const` è la **dimensione** dell'array

4.2.2 semantica

- viene allocato spazio in memoria per contenere l'array
- ogni elemento **contiene** un dato del tipo base
- la dimensione `int_const` è **un'espressione costante** di tipo `int` (≥ 0)

4.3 Accesso agli elementi

Ogni elemento dell'array **contiene** un dato del tipo di base, per **accedere ai singoli elementi** si utilizza la notazione:

id`[0]`, id`[1]`, ..., id`[int_const - 1]`

Il contenuto delle parentesi `[.]` è detto **indice**. Un inidice si dice **valido** se assume un valore compreso tra `0` e `int_const - 1` (dimensione dell'array - 1)

4.3.1 Assegnamenti

Per assegnare un valore ad elemento di un array si utilizza l'operazione di assegnamento

```
int n = 2;
a[n + 2] = 35;
```

Assegna all'elemento `A[4]` il valore `35`

4.3.2 Accesso sequenziale agli array

Spesso è necessario elaborare gli elementi di una array in sequenza, partendo dal primo elemento.

Solitamente si utilizza un ciclo `for`, la cui variabile di controllo (`i`) viene usata come indice dell'array
Esempi:

- inizializzazione del contenuto dell'array

```
int A[lenght];
for(int i = 0; i < lenght; i++){
      cin>> A[i];
}
```

- somma degli elementi

```
int sum = 0;
for int i = 0; i < lenght; i++){
      sum = sum + A[i];
}
```

4.3.3 Errori

- La lunghezza dell'array **NON PUÒ ESSERE** variabile

```
cout<< "lunghezza dell'array";
cin >> lenght;
int A[lenght];
```

- accesso a elementi al di fuori della lunghezza (**out of range**)

```
const int lenght = 10;
for int i = 0; i < lenght; i++){
      A[i] = i*i
}
```

4.4 Array come argomento di funzioni

Un parametro formale **può essere un array** però l'**array viene sempre passato per riferimento**.

Un parametro formale di tipo array viene specificato utilizzando le parentesi quadre "[]" **senza alcun indice**

esempio:

```
int function (int vec[], int n){
      ...
}
```

Il parametro attuale di tipo array è specificato **solamente** dall'identificatore (senza utilizzare le `[]`) **esempio:**

```
int main(){
  ...
 function(a, lenght);
  ...
}
```

4.4.1 const

Quando una funzione non modifica un array (accede all'array in lettura) conviene passare l'array in modalità `const`

Esempio:

```
1  void add_arrays(const int A[], const int B[], int SUM[], int n){
2      int i;
3      for (i = 0; i < n; i = i+1) SUM[i] = A[i] + B[i] ;
4  }
5  int main(){
6      const int length = 100 ;
7      int vec1[length], vec2[length], vecsum[length], i ;
8      for (i=0 ; i < length; i=i+1){
9          vec1[i] = 2*i ; vec2[i] = 2*i+1 ;
10     }
11     add_arrays(vec1,vec2,vecsum,length) ;
12     return(0) ;
13 }
```

Cosa succede se si rimpiazza la linea `11` con: `add_arrays(vec1,vec2,vec1,length);`?

4.4.2 Esercizi:

1. scrivere una funzioone che dato un numero binario memorizzato in un array lo converte in decimale
2. *palindrome:* un array di caratteri è palindromo se leggendolo da destra verso sinistra o da sinistra verso destra si ottiene lo stesso array.
   Scrivere una funzione che verifica se un array è palindromo o meno
3. scrivere una funzione che prende in input un array di interi `a` e un intero `n` e ritorna `true` o `false` a seconda che `n` si trovi in `a` oppure no
4. scrivere una funzione che prende in input un array di interi e ne stampa gli elementi senza stampare duplicati

4.5 Ricerca di elementi in un array

Cerchiamo di verificare se un elemento `k` è presente o meno in un array

L'algoritmo:

1. utilizzare un ciclo per esaminare gli elementi dell'array uno alla volta confrontandoli con `k`
2. uscire dal ciclo una volta trovato un valore uguale a `k`
3. utilizzare una variabile `bool` per indicare che il valore è stato trovato e si può uscire dal ciclo (comando `while`)
4. la funzione ritorna l'indice dell'elemento
5. costo computazionale nel caso pessimo, ovvero quando l'elemento non è presente: `lenght_array`

```
int search(int A[], int lenght, int k){
    bool found = false;
    int i = 0;
    while (!found && (i<lenght)){
        if (A[i] == k) found = true;
        else i = i + 1;
    }
    return(i);
}
```

4.6 Ordinamento di un array

Ora, invece, vogliamo **ordinare** una lista di valori.

È una operazione molto comune

Ci sono molti algoritmi di ordinamento

- alcuni sono semplici da comprendere
- altri sono molto efficienti computazionalmente

Quando l'ordinamento dell'array `A` sarà completato si avrà un array in cui:

$$\texttt{A[0]} \leq \texttt{A[1]} \leq \ldots \leq \texttt{A[lenght-1]}$$

4.6.1 Selection sort

Algoritmo:

1. ricerca del più piccolo valore nell'array `A` e sia `i` la sua posizione
2. sostituisci `A[0]` con `A[i]`
3. iniziando da `A[1]` ricerca il più piccolo valore nell'array e sostituiscilo con `A[1]`
4. iniziando da `A[2]` ricerca il più piccolo valore nell'array e sostituiscilo con `A[2]`
5. . . .

```
void selection_sort(int A[], int lenght){
    int min;
    for(int i = 0; i < (lenght - 1); i++){
        min = i;
        for(int j = i + 1; j < lenght; j++){
            if(A[min] > A[j]) min = j;
            swap(A[i], A[min]);
        }
    }
}
```

4.6.2 Bubble sort

Schema del boubble sort:

```
3    10    9    2    5
^
```

non c'è scambio perché $3 \leq 10$

```
3    10    9    2    5
     ^
```

c'è scambio perché `10 > 9`

E così via finché l'array non viene ordinato

```
void bubble_sort(int A[], int lenght){
    for(int i = 0; i < lenght; i++){
        for(int j = 0; j < (lenght - 1 - i); j++){
            if (A[j] > A[j+1]){
                scambia (A[j], A[j+1]);
            }
        }
    }
}
```

4.7 Algoritmo di ricerca su array ordinati

Per chercare se un array `A` i cui elementi sono **ordinati** contiene o meno un elemento `k` si utilizza **la ricerca binaria**

1. si accede all'elemento memorizzato a **metà** di `A` e si verifica se esso è uguale a `k`
   - se è uguale a `k` l'algoritmo termina e ritorna l'indice
   - altrimenti sceglie la metà appropriata di `A` (maggiore o minore di `k`) e ripete il passo 1.
2. l'algoritmo termina con `-1` se `k` non è presente

Implementazione:

```
int bin_search(int A[], int lenght, int k){
    bool found = false;
    int l = 0;
    int r = lenght;
    int m;
    while (!found && (l < r)){
        m = (r + l)/2;
        if (A[m] == k) found = true;
        else if (A[m] > k) r = m;
        else l = m + 1;
    }
    if (found) return(m);
    else return(-1);
}
```

4.8 Esercizi

1. definire una funzione che dato un array restituisce la posizione della seconda occorrenza del primo carattere che occorre almeno due volte, restituisce `-1` se nessun carattere occorre almeno due volte.

2. scrivere un programma che implementa una pila utilizzando un array (definire le funzioni `is_empty`, `push`, `pop`; la funzione `push` ritorna `overflow` se l'array è pieno

3. scrivere un programma che implementa due pile utilizzando un solo array.
   In particolare, dato un array di lunghezza `L` definire le funzioni `push1`, `pop1`, `is_empty1` e le funzioni `push2`, `pop2`, `is_empty2` che implementano sull'array le note funzioni sulle pile.
   Le funzioni `push1` e `push2` restituiscono `overflow` solamente se l'array è pieno.

4. Definire una funzione `void parola(char str[], int n, char dest[])` che prende come parametri un array `str` e un intero `n` e restituisce nel parametro `dest` la parola corrispondente all'ennesima parola dentro `str`.
   Si assuma che una parola sia una qualunque sequenza di caratteri diversi da spazio e che le parole siano separate tra loro da uno o più spazi.
   In caso di errore la funzione restituisce la parola vuota.

Chapter 5

Stringhe

Per rappresentare e memorizzare caratteri alfanumerici in C++ abbiamo **2 opzioni**:

1. Vettori di caratteri
2. la classe standard `string`

Il prof ha deciso che verrà trattato **solamente il 1. modo** e che **il secondo non si potrà usare nelle prove d'esame**

5.1 Array di char

Le sequenze di caratteri sono dette **stringhe**

Le stringhe non sono un nuovo tipo di dato

Le stringhe sono **implementate** come **array di caratteri**

5.1.1 Dichiarazione

Per dichiarare una variabile di tipo stringa si utilizza il pattern:

```
char Nome_array[Dimesione_max + 1];
```

- può memorizzare stringhe di massimo **9** caratteri
- l'ultimo carattere è il **carattere null** `'\0'` che indica la fine della stringa
- il carattere null è un carattere singolo
- un array di caratteri che non contiene `'\0'` **non è significativo**
- per contenere una stringa è **necessario sovrastimare** l'array
- bisogna far attenzione a non uscire dalla dimensione dell'array con l'input

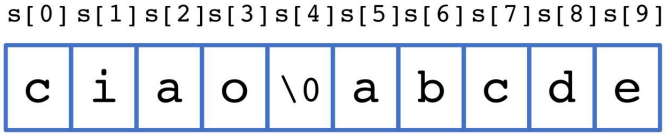
Dichiarare una stringa come `char s[10]` crea spazio per solamente `9` caratteri, il carattere `'\0'` **è necessario** e richiede una posizione

Ogni carattere dopo il null **non è significativo**

Se il primo carattere della stringa è `\0` allora la stringa è vuota

La presenza di `'\0'` **consente di determinare la lunghezza** della stringa

Esempio:



5.1.2 Inizializzazione

Per inizializzare una variabile di tipo stringa durante la dichiarazione:

```
char my_message[20] = "Hi there.";
```

- `'\0'` aggiunto automaticamente
- si può usare in alternativa

```
char short_string[ ] = "abc";
```

La lunghezza è determinata automaticamente: in questo caso è `4`

5.2 Operazioni su stringhe

5.2.1 strlen

`strlen` ritorna il numero di caratteri nell'argomento

```
int x = strlen(a_string);
```

Non conta il carattere `\0`

5.2.2 strcat

`strcat(a_string, b_string)` concatena due stringhe.

- Il secondo argomento è giustapposto al primo
- il risultato viene memorizzato nel primo argomento

esempio:

```
char string_var[20] = "The rain";
strcat(string_var, "in Spain");
```

Ora `string_var` contiene `"The rainin Spain"`

5.2.3 strncat

`strncat(a_string, b_string, n)` concatena il primo e i primi `n` caratteri del secondo argomento.

Il terzo parametro specifica il numero di caratteri da concatenare (il secondo argomento viene troncato a quei caratteri)

```
char a_string[20] = "The rain";
strncat(a_string, " in Spain", 11);
strncat(a_string, "and in Italy", 5);
```

ora `a_string` contiene `"The rain in Spainand i"`

5.2.4 strncpy

Il comando di assegnamento

```
char a_string[20] = "hello";
```

è **illegale** perché il comando di assegnamento non opera correttamente con le stringhe

Il metodo standard è usare la funzione di libreria `strncpy` definita in `cstring`

```
#include <cstring>
. . .
char a_string[lenght];
strncpy (a_string, "Hello", size);
```

Mette **Hello** seguito dal carattere `\0` in `a_string` se `size ¡ lenght` altrimenti non lo inserisce e il risultato non è una stringa!

5.2.5 strcpy

`strcpy` è un'alternativa alla precedente `strncpy`

Questa funzione:

- prende in input 2 stringhe
- **non verifica** la lunghezza della stringa del primo argomento, quindi può tentare di scrivere **oltre la lunghezza dichiarata**

5.2.6 strcmp

L'uguaglianza tra stringhe **non si esprime con** "==", si utilizza la funzione `strcmp` per confrontare variabili di tipo stringa.

esempio

```
#include <cstring>
. . .
if (strcmp(a_string, b_string))
    cout << "strings are not the same.";
else cout<< "strings are the same;
```

"`strcmp`" confronta i codici numerici dei **caratteri corrispondenti** nelle due stringhe

Se le fue strenghe sono le stesse allora `strcmp` ritorna `0` (`false`)

Al primo carattere differente

- `strcmp` ritorna un **valore negativo** se il codice del carattere del primo parametro **è minore**
- `strcmp` ritorna un **valore positivo** se il codice del carattere del primo parametro **è maggiore**
- i valori non-zero sono interpretati come `true`

5.3 Output di stringhe

Le stringhe possono essere date in output:

```
char news[20] = "stringhe,";
cout<< news << " Wow." <<endl;
```

output:

```
> stringhe, Wow.
```

5.4 Input di stringhe

È possibile fare input di stringhe con "`cin >>`"

L'input **termina con uno spazio bianco**, il carattere '`\0`' viene aggiunto automaticamente
Esempio:

```
char a[80], b[80];
cout<< "Enter input: " <<endl;
cin >> a >> b;
cout<< a << b <<"End of Output;
```

con input "`un buon pomeriggio`" stamperà:

```
> Enter input:
> unbuonEnd of Output
```

5.4.1 Leggere una intera riga: getline

Per leggere una riga intera si utilizza la funzione `cin.getline`.

`cin.getline` legge una riga intera inclusi gli spazi.

`cin.getline` ha **due** argomenti:

- il **primo** è la variabile di tipo stringa che riceverà l'input
- il **secondo** è un intero (solitamente la lunghezza dell'array passato come primo argomento) che determina **il massimo numero di caratteri** preso in input e che sarà memorizzato nella stringa

Sintassi:

```
cin.getline(A, m+1);
```

`cin.getline` termina di leggere quando ha raggiunto il **numero di caratteri meno uno** specificato nel secondo argomento

5.5 Dalle stringhe ai numeri

Le funzioni di conversione

- `atoi : string` $\rightarrow$ `int`
- `atol : string` $\rightarrow$ `ling int`
- `atof : string` $\rightarrow$ `double`

Si trovano nella libreria `cstdlib`

Per usarle:

```
#include <cstdlib>
```

5.5.1 Dalle stringhe agli interi

Quando si fa input di interi **è spesso conveniente** leggere l'input come una stringa e poi **convertire** la stringa in intero

- perché quando si legge una quantità di denaro c'è spesso "`€`" o "`$`"
- perché quando si legge una percentuale c'è sempre il simbolo "`%`"

Per leggere un intero come sequenza di caratteri si legge l'input di una variabile tipo stringa, si rimuovono i caratteri indesiderati che si trovano in testa (perché `atoi` termina appena trova un carattere non cifra) ed infine si utilizza `atoi` per convertire la stringa in intero

5.5.2 Dalle stringhe ai long int

Le stringhe di cifre **più grandi** possono essere **convertiti** con la funzione `atol`.

- `atol` ritorna un `long int`
- ci sono le stesse problematiche di `atoi` per i caratteri non cifra

5.5.3 Dalle stringhe ai double

Una stringa può essere convertita in tipo `double` con la funzione `atof`

- `atof("9.99")` ritorna `9.99`
- `atof("$9.99")` ritorna `0.0` perché `$` non è una cifra

5.6 Stringhe come argomenti di funzioni

- Quando le stringhe sono parametri formali o attuali di funzioni vengono **considerati come array**
- Se una funzione **cambia valore** di un parametro stringa allora è bene passare la lunghezza, per evitare eccessi out-of-bound.
- Se una funzione **non cambia il valore** di un parametro stringa allora, non è necessario passargli la lunghezza, perché è determinata da "`\0`"

5.7 Esercizi:

1. Scrivere una funzione che concatena due stringhe, mettendo il risultato nella prima
2. Scrivere una funzione che verifica se una stringa è composta da sole lettere maiuscole
3. scrivere una funzione che elimina da una stringa tutti i caratteri dopo l'ennesimo

Chapter 6

Strutture

Una **Struttura** è un dato composto da elementi generalmente differenti tra loro (eterogenei) raggruppati sotto un unico nome.

Utilizziamo le strutture per creare dei tipi di dati che non esistono di dafault.

Le strutture vengono anche chiamate **record**.

Anche gli array sono raggruppamenti di dati, **ma dello stesso tipo**.

Esempi:

giorno	19	12	2001	studente	Marco	Bianchi	Unibo
	`giorno`	`mese`	`anno`		`nome`	`cognome`	`istituto`

6.1 Dichiarazioni di strutture

In `C++` è possibile definire tipi di dato che raggruppano in una univa struttura informazioni di tipo diverso.
Occorre specificare:

- il nome di ciascuna componente (i componenti sono detti **campi**)
- il **tipo** di informazione memorizzato nelle componenti

Esempi:

```
struct data{
    int giorno;
    int mese;
    int anno;
};
```

```
struct studente{
    char nome[10];
    char cognome[10];
    char istituto[10];
    struct data data_di_nascita;
};
```

Sintassi:

```
struct id{
    tipo_0 id_list;
    tipo_1 id_list;
    ...
    tipo_n id_list;
};
```

- l'identificatore `id` è il **nome del tipo** della struttura
- ogni `id_list` è un elenco di uno o più nomi di componenti separati da virgole ”,”
- il **tipo** di ogni componente è un tipo semplice o un tipo precedentemente definito (può derivare da altre strutture)

È importante notare che la **dichiarazione di una struttura** serve a quantificare lo spazio di memoria necessario per definire le variabili di quel tipo.

Una volta definita una struttura (come nei 2 esempi di sopra) è possibile dichiarare variabili di quel tipo:

```
int main(){
    data oggi, esame;
    studente Turro, Pippo;

    return 0;
}
```

6.2 Operazione di selezione e i campi

Ora vogliamo accedere i campi di una struttura per poter analizzare/lavorare su di essi.

Per accedere ai campi si utilizza la *dot-notation* dopo l'identificatore si mette un punto e il nome del campo desiderato.

Per esempio dichiarando così:

`oggi.anno = 2022;`	mi darà: `2022`
`oggi.giorno = 23;`	mi darà: `23`

6.2.1 Operazione di copia

È possibile copiare e assegnare strutture:

`Marco_Rossi = Pinco_Pallino`

Facendo così tutti i campi di `Pinco_Pallino` vengono copiati nei corrispondenti campi di `Marco_Rossi`.

Anche quando i campi sono array (`nome`, `cognome`, ...)

NON è però possibile **confrontare due strutture**

6.2.2 Inizializzazione

Dopo aver dichiarato una struttura (es. `data`)

È possibile dichiarare ed inizializzare una variabile in questo modo:

`data oggi = {23, 05, 2022};`

Risultato:

I campi `giorno`, `mese`, `anno` di `oggi` verranno inizializzati rispettivamente a `23, 05, 2022`

6.3 Strutture passate come parametri

6.3.1 Strutture come parametri di funzioni

Le strutture possono essere passati come argomenti di funzioni ed essere restituite come i valori.

Esempio:

```
struct point{
    int x;
    int y;
};

point make_point(int a, int b){
    point p;
    p.x = a;
    p.y = b;
    return(p);
}
```

Per passare le strutture serve un **passaggio per valore**

```
void incr_point(point& a){
    a.x = a.x + 1;
    a.y = a.y + 1;
};
    ...
    point b;
    b.x = 1;
    b.y = 2;
    incr_point(b);
    cout<<b.x <<" " <<b.y;
    ...
```

```
Output: 2 3
```

6.4 Esercizi Svolti

Vediamo ora alcuni esercizi svolti per poter dare un'idea dello stile di codice utilizzato:

6.4.1 Il tipo di dato Pila

- Implementare il tipo di dato pila di interi (LIFO) (per mezzo di strutture e array)
- l'implementazione deve fornire le operazioni:
  - `creazione di una pila vuota`
  - `test per pila vuota`
  - `inserimento`
  - `estrazione`

```
const int lenght = 1000;

struct stack{
    int data[lenght];
    int top;
};

stack new_stack(){
    stack tmp;
    tmp.top = lenght;
    return tmp;
}

bool is_empty(stack s){
    return(s.top == lenght);
}

stack push(stack s, int e){
    if (s.top == 0){
        cout<<"Error, stack full
    }
}

stack pop(stack s){
    if (s.top == lenght){
        cout<<"ERROR, EMPTY STACK\n";
    }else{
        s.top = s.top + 1;
    }
    return(s);
};
```

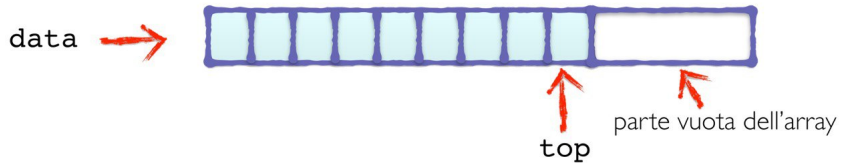
Problemi:

1. Le risposte erronee sul terminale non ci dovrebbero essere (bisognerebbe prevenire a monte questi errori).
2. Non c'è *incapsulamento*: la struttura degli elementi è visibile e modificabile dall'utente senza usare le funzioni.

6.4.2 Tipo di dato Insieme

Implementare il tipo di dato insieme di interi (per mezzo di strutture e array)

- l'implementazione deve fornire le seguenti operazioni:
  - `creazione di un insieme vuoto`
  - `appartenenza di un elemento all'insieme`
  - `intersezione`
  - `unione`
- implementiamo un insieme attraverso una struttura on due campi:
  - un campo `data` che è un array
  - un campo `top` che contiene l'indice dell'ultimo elemento valido nell'array.




```
#include <iostream>
using namespace std;

const int lenght = 1000;

struct set{
    int data[lenght];
    int top;
};

set empty_set(){
    set tmp;         // non esiste un elemento in data[-1]
    tmp.top = -1;    // se top == -1 significa che l'insieme
    return(tmp);     // e' vuoto
};

bool is_in(set s, int e){
    if (s.top == -1) return(false);
    else{
        int l = 0;
        int r = s.top;
        int m;
        bool found = false;
        while ((l <= r) && !found){      // algoritmo di ricerca binaria
            m = (l + r)/2;               // perche l'array e' ordinato
            if (s.data[m] == e) found = true;
            else if (s.data[m] > e) r = m - 1;
            else l = m + 1;
        }
        return(found);
    }
}

set intersection(set s1, set s2){
    if ((s1.top == -1) || (s2.top == -1)) return(empty_set());
    else{
        set s;
        int i = 0;
        int j = 0;
```

```
40          s = empty_set();
41          while ((i <= s1.top) && (j <= s2.top)){
42              if (s1.data[i] == s2.data[j]){
43                  s.top = s.top + 1;
44                  s.data[s.top] = s1.data[i];
45                  i = i + 1;
46                  j = j + 1;
47              }else if (s1.data[i] < s2.data[j]) i = i + 1;
48              else j = j + 1;             // trascrizione di due array ordinati
49          }                               // in un terzo array (ordinato)
50          return(s);
51      }
52  }
53
54  set set_union(set s1, set s2){
55      if (s1.top == -1) return(s2);
56      else if (s2.top == -1) return(s1);
57      else{
58          set s = empty_set();
59          int i1 = 0;
60          int i2 = 0;
61          while ((i1 <= s1.top) && (i2 <= s2.top) && (s.top < lenght)){
62              if (s1.data[i1] < s2.data[i2]){
63                  s.top = s.top + 1;
64                  s.data[s.top] = s1.data[i1];
65                  i1 = i1 + 1;
66              }else if (s1.data[i1] > s2.data[i2]){
67                  s.top = s.top + 1;
68                  s.data[s.top] = s2.data[i2];
69                  i2 = i2 + 1;
70              }else{                      // unione di due array ordinati
71                  i1 = i1 + 1;
72                  i2 = i2 + 1;
73              }
74          }
75          while ((i1 <= s1.top) && (s.top < lenght)){
76              s.top = s.top + 1;
77              s.data[s.top] = s1.data[i1];
78              i1 = i1 + 1;
79          }
80          while ((i2 <= s2.top) && (s.top < lenght)){
81              s.top = s.top + 1;
82              s.data[s.top] = s2.data[i2];
83              i2 =i2 + 1;
84          }                               // copia delle parti restanti
85          if ((i1 <= s1.top) || (i2 <= s2.top)){
86              cout<<"write error" <<endl;
87          }
88          return(s);
89      }
90  }
```

6.5 Esercizi

1. scrivere un programma che definisce una `struct` studente, chiede all'utente di inserire i dati di uno studente e stampa poi il nome dello studente e la media dei suoi voti.
2. scrivere un programma che definisce una struttura giorno dell'anno, chiede all'utente di inserire un giorno e calcola quanti giorni sono passati dall'inizio dell'anno
3. scrivere un programma che gestisce gli studenti di informatica, con le seguenti funzioni:
   (a) inserimento di uno studente
   (b) inserimento di un esame
   (c) calcolo della media dei voti di uno studente
   (d) calcolo dell'età media degli studenti
4. scrivere un programma che funziona da agenda telefonica, con le seguenti funzioni:
   (a) inserimento di una persona nell'agenda
   (b) ricerca di una persona per nome
   (c) cancellazione di una persona

Chapter 7

Puntatori

I puntatori sono **il tipo di dato degli indirizzi di memoria**

È un concetto complesso **perché non esiste in matematica**. Gli indirizzi di memoria possono essere usati in alternativa alle variabili, se una variabile è memorizzata in 4 locazioni di memoria consecutive allora l'indirizzo di memoria della prima locazione può essere utilizzato in alternativa alla variabile.

È un meccanismo che già avviene con il *passaggio per riferimento* in una dichiarazione di funzione, quando la funzione viene invocata si passa l'indirizzo del parametro attuale.

7.1 Dichiarazioni di puntatori

I linguaggi di programmazione consentono di definire tipi di dato **indirizzi di memoria**.

È quindi possibile definire e creare strutture di dati di dimensioni non note staticamente, ovvero **definire VARIABILI DINAMICHE**.
Per indicare che un identificatore è un puntatore a valori di un certo tipo, nella dichiarazione, si mette prima dell'identificatore "`*`".

Esempi:

```
int *p;              // p e' un puntatore a interi
double *f_p;         // f_p e' un puntatore a double
int *q, n;           // q e' un puntatore a interi
                     // n e' un intero
```

7.2 Le operazioni su puntatori

- `NULL` puntatore vuoto
- `new` allocazione blocco di memoria
- "`*`" dereferenziazione
- "`&`" indirizzo di
- `delete` deallocazione blocco di memoria

7.2.1 NULL

La costante `NULL` rappresenta il **puntatore vuoto**
Può essere assegnata **solamente** ad un puntatore

Esempi:

```
int *p;
char *q, r;
p = NULL;
q = NULL;
if (p == NULL) ...
if (p == q) ...          // ERRORE! 'p' punta a int e 'q' punta  a char
r = NULL;                // ERRORE! 'r' non e' un puntatore
```

7.2.2 new

La funzione `new` permette di allocare dinamicamente la memoria per una variabile di tipo puntatore.

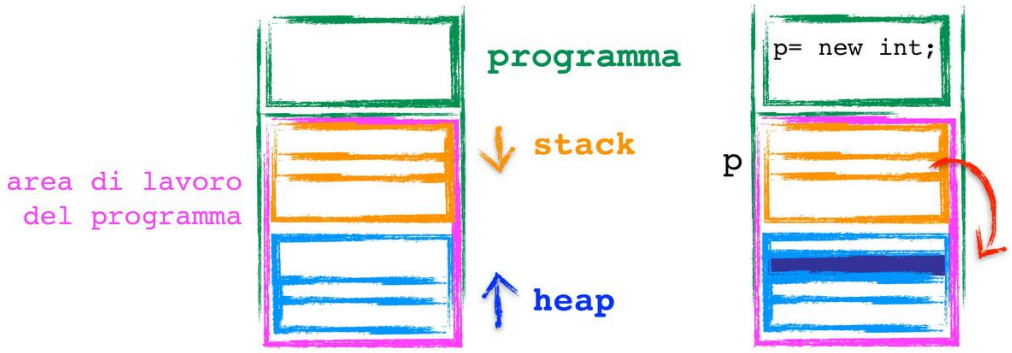
`new` prende in input un tipo e restituisce un puntatore ad un **nuovo** blocco di memoria che contiene elementi di quel tipo.

Facciamo ora un attimo di precisazione sul funzionamento della memoria (ram) del programma.

La ram del programma è divisa in due parti:

stack e **heap**

- **heap** è l'area di memoria in cui sono allocati nuovi blocchi di memoria (ad esempio dalla funzione `new`)
- **stack** è l'area di memoria in cui venfono allocati i **record di attivazione** delle funzioni.



Le celle di memoria allocate con l'operatore `new` sono chiamate **variabili dinamiche**.

L'area di memoria per le variabili dinamiche viene creata e distrutta durante l'esecuzione del programma.

L'area di memoria per le altre variabili è nota a tempo di compilazione e creata quando il programma o la funzione vine invocata e distrutta quando il programma o la funzione termina.

7.2.3 *

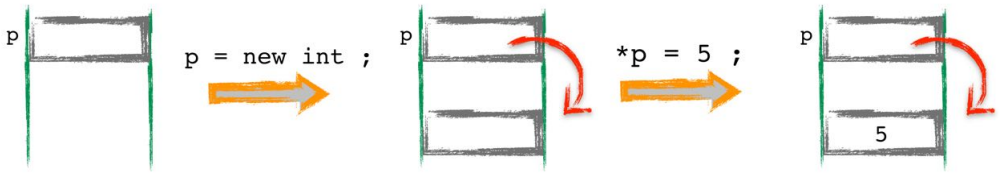
Se si esegue:

```
1 int *p;
2 p = new int;
```

Come si fa ad accedere alla cella puntata di `p`? (`p` è stata allocata dinamicamente ln 2)

Serve un'operazione che, preso un puntatore, restituisce l'indirizzo di memoria per accedere all'elemento puntato da esso.

Per questo si utilizza l'operazione di **dereferenziazione** "`*`": `*p = 5;`

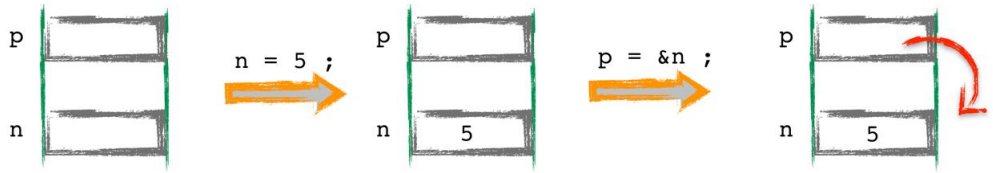


7.2.4 &

L'operatore &, applicato ad una espressione restituisce il suo indirizzo di memoria (che può essere assegnato ad un puntatore)

Esempio:

```
1  int *p, n;
2  n = 5;
3  p = &n;
```



È ERRORE FARE: `p = &5` oppure `p = &(n*5)`

7.2.5 delete

`delete` libera l'area di memoria puntata dall'argomento (l'argomento deve essere un puntatore).

Quest'area potrà essere riutilizzata da successive chiamate a `new`

`delete` non fa nulla se il puntatore in input punta a `NULL`

Esempio:

7.2.6 Dangling pointers

Dopo l'invocazione di `delete` il valore del puntatore p indefinito diventa quindi un **danglilng pointer**.

Regola di programmazione: riassegnare il puntatore dopo l'invocazione di `delete`

Esempio:

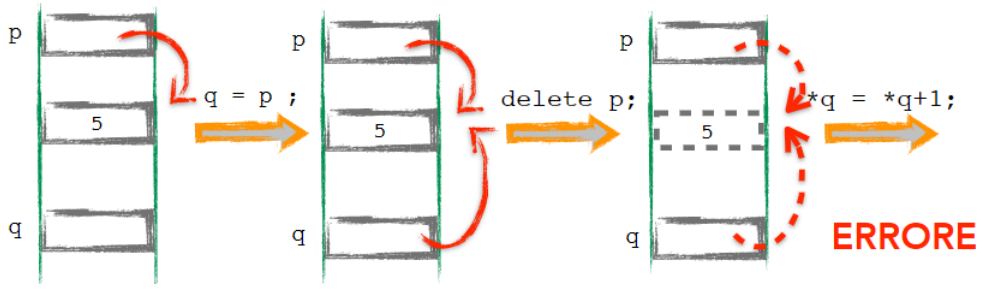
```
delete p;
p = NULL;
```

Problemi:

Se un altro puntatore punta alla stessa area di memoria anche il suo valore è indefinito.

Esempio:

```
q = p;
delete p;
*q = *q + 1;
```

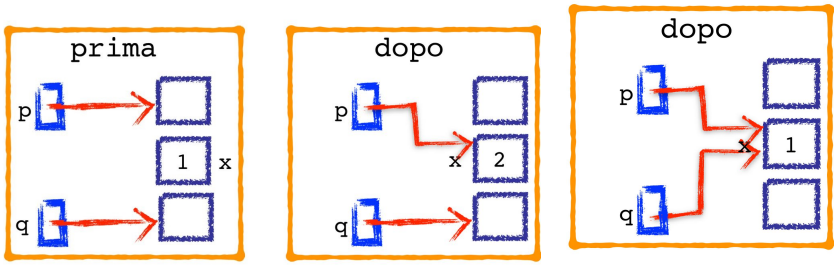


7.2.7 Aliasing

L'*Aliasing* indica la situazione in cui una stessa posizione di memoria è associata a nomi simbolici diversi all'interno di un programma.

Banalmente due puntatori puntano alla stessa cella.

```
// prima
int *p, *q, x;
x = 1;
// ==============
p = &x;
*p = 2;
// ==============
q = p;
*q = 1;
```



7.2.8 Definizione di nuovi tipi

È possibile definire nuovi tipi ed assengargli un nome

- la parola chiave è `typedef`

Sintassi:

```
    typedef Tipo_Noto identificatore;
```

`Tipo_Noto` è un tipo già definito

La definizione di nuovi tipi ci torna comoda per definire **tipi di puntatori**:

Esempio:

```
typedef int *p_int;
```

- definisce un tipo "**puntatore a interi**"
- è possibile dichiarare variabili di tipo `p_int`

`p_int p;;` è equivalente a `int *p`

Evita confusioni tra puntatori a interi e variabili intere nelle dichiarazioni

```
int *p, *q, x, y
```

```
p_int p, q;
int x, y;
```

7.3 Esempi/Esercizi

1. definire una variabile di tipo intero, incrementarla due volte attraverso due puntatori distinti e stampare il risultato
2. allocare una variabile di tipo intero, incrementarla attraverso un puntatore, stampare il risultato e deallocarla
3. predere in input 10 interi e memorizzarli in un array di 10 interi utilizzando i puntatori. Poi stampare i valori
4. definire una struttura di 5 campi interi e memorizzarci 5 interi presi in input utilizzando i puntatori. Poi stampare i valori
5. definire una variante della funzione "scambia" che scambia i valori di due variabili utilizzando i puntatori e usarla all'interno del main
6. ridefinire un algoritmo di ordinamento di array accedendo agli elementi attraverso puntatori.

7.4 Puntatori passati come parametri

Dobbiamo distinguere:

- il **parametro formale** è di tipo puntatore
- il **parametro attuale** è l'indirizzo di una variabile

Il passaggio dei puntatori è **per valore**: vengono copiati gli indirizzi che consentono di modificare le variabili chiamate.

Esempio: la funzione `scambia`

Siano `a` e `b` di tipo `int`

```
void scambia(int x, int y){
    int tmp;
    tmp = x;
    x = y;
    y = tmp;
}   // invocata con scambia(a, b)
```

```
void scambia(int& x, int& y){
    int tmp;
    tmp = x;
    x = y;
    y = tmp;
}   // invocata con scambia(a, b)
```

```
void scambia(int *x, int *y){
    int tmp;
    tmp = *x;
    *x = *y;
    *y = tmp;
}   // invocata con scambia(&a, &b)
```

```
void scambia(int *x, int *y){
    int *tmp;
    tmo = x;
    x = y;
    y = tmp:
}   // invocata con scambia(&a, &b)
```

7.5 Le strutture dati dinamiche

Le **strutture dati dinamiche** sono strutture dati le cui dimensioni possono essere estese o ridotte durante l'esecuzione del programma.
Notiamo che tutti i tipi studiati finora hanno una dimensione che è nota **staticamente** (ovvero viene definita e non può essere modificata a seconda delle necessità)

7.5.1 Le liste

Una lista è una sequenza di nodi che contengono valori (di uno o più tipi) e in cui ogni nodo, a parte l'ultimo, è collegato al nodo successivo **con un puntatore**.

Una lista, di solito, è una struttura con un campo che contiene un puntatore allo stesso tipo della struttura.

Un buon modo per *visualizzare* una lista è attraverso nodi e frecce che li connettono.

7.5.2 I nodi

I rettangoli **blu** del disegno rappresentano i nodi della lista

- i nodi contengono **le informazioni** memorizzate nella lista e **un puntatore**
- il puntatore punta **all'intero nodo**, non ad una parte di esso
- il puntatore  rappresenta `NULL`

7.5.3 Implementazione dei nodi

I nodi vengono implementati in C++ mediate il tipo di dato `struct` o mediante le classi.
Esempio:
Una struttura che memorizza 2 elementi, un intero e un puntatore ai nodi della struttura.

```
24 struct node{
25     int val;
26     node *next;
27 };
```

Questa definizione circolare, un puntatore che punta allo stesso tipo dal quale è originato, è ammessa in C++.

7.5.4 La testa della lista

Il rettangolo chiamato `head` **non è un nodo della lista** ma un puntatore alla lista.
`head` è dichiarato

```
node *head;
```

Oppure è possibile definire un tipo di dato *puntatore alla lista* e dichiararlo di conseguenza

```
typedef node* p_lista;
p_lista head;
```

7.5.5 Accesso agli elementi della lista

Vogliamo adesso un modo per modificare l'intero nel primo nodo da `13` a `5`.

```
(*head).val = 5;
```

- `head` è di tipo puntatore, quindi `*head` restituisce l'oggetto puntato (il 1° nodo)
- l'oggetto puntato è una `struct`, si utilizza quindi la **dot-notation** per accedere al campo `val`
- le parentesi sono necessarie perché il "`.`" ha la precedenza sul "`*`"

L'operatore freccia

L'operatore `->` combina le azione della dereferenziazione e dell'accesso a un campo di una struttura, quindi:

```
(*head).val = 5;
```

Può essere riscritto

```
head->val = 5;
```

7.5.6 Come creare una lista

Iniziamo con la definizione

```
struct lista{
    int val;
    lista *next;
};
typedef lista *p_lista;
```

Definiamo la testa della lista:
`p_lista head;`
Creiamo il primo nodo
`head = new lista;`

Ore che `head` punta ad un nodo occorre inizializzare i campi con dei valori

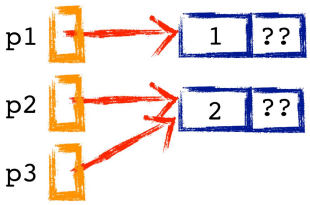
```
head->val = 5;
head->next = NULL;
```

Dato che questo è l'unico nodo, il puntatore al resto della lista è inizializzato a `NULL`.

7.6 Operazioni su lista

7.6.1 Dichiarazioni e inizializzazioni

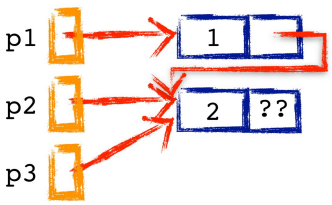
```
p_lista p1, p2, p3;
p1 =  new lista;
p1->val = 1;
p2 = new lista;
p2->val = 2;
p3 = p2
```



Collegamento tra nodi

```
p1->next = p2;
```

Scrivendo così possiamo far puntare il primo nodo al secondo.



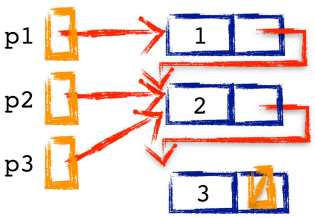
Ora abbiamo **3** modi per accedere al campo `val` del secondo nodo (quello con 2)

`p2->val`	oppure	`(*p2).val`
`p3->val`	oppure	`(*p3).val`
`(p1->val)->val`	oppure	`(*((*p1).next)).val`

7.6.2 Aggiunta in coda

```
p2->next = new lista;
(p2->next)->val = 3;
(p2->next)->next = NULL;
```

La **testa** è il primo elemento della lista ed è puntata da `p1`, una funzione che conosce l'indirizzo contenuto in `p1` **ha accesso ad ogni elemento della lista**.



7.6.3 Inserimento in testa

Progettiamo una funzione che prende una lista ed un elemento e ritorna la lista con il nuovo elemento in testa

`p_lista head_insert(p_lista head, int el);`

`head_insert` creerà un nuovo nodo che conterrà `el` nel campo `val`.

Dobbiamo:

- creare un nuovo **nodo** puntato da `tmp_head`
- inizializzare il nuovo nodo `*temp_head` con `el` e con `head`
- ritornare il puntatore `tmp_head`

```
1 p_lista head_insert(p_lista h, int el)
2 {
3     p_lista = tmp_head;
4     tmp_head = new lista;
5     tmp_head->val = el;
6     tmp_head->next = h;
7     return(tmp_head);
8 }
```

7.6.4 Inserimento di un nodo in una posizione specifica

Vediamo ora come inserire un nodo **in un posto specifico** di nostro interesse.

`void insert(p_lista prev_node, int val);`

Ci potrebbe venir chiesto di riordinare una lista in ordine crescente/decrescente o in generale spostare un nodo tra altri due.

Per visualizzare meglio questo concetto abbastanza complicato possiamo immaginarci un vettore;
Vogliamo inserire `val` nella 4 cella, (passiamo quindi nella funzione il puntatore alla cella `3` e `val`)
Dobbiamo:

1. Controllare se il nodo in input è l'ultimo (ln `3`)
2. creare un nuovo nodo "`new_node`" (ln `4`)
3. inserire i dati `val` in `new_node` (ln `5`)
4. inizializzare il puntatore (`next`) del nuovo nodo al nodo successivo di quello preso in input (ln `6`)
5. far puntare il nodo preso in input al nuovo nodo creato (ln `7`)

```
1 void insert(p_lista prev_node, int val)
2 {
3     if (prev_node != NULL){
4         p_lista new_node = new node;
5         new_lista->data = val;
6         new_node->next = prev_node->next;
7         prev_node->next = new_node;
8     }
9 }
```

7.6.5 Ricerca di elementi in una lista

`p_lista search_el(p_lista p, const int e);`

poiché il puntatore `p` è passato per valore, possiamo scorrere la lista utilizzando `p`.

L'algoritmo che adottiamo consiste nello scorrere tutta la lista puntata da `p` finché non si trova l'elemento `e` oppure il valore `p` è `NULL`.

Qua poi il prof. fa una gran sbatta di esempi vedendo gli errori, se vi va leggetevela nelle slide, io riporto solo la funzione finita e commentata.

```
p_lista search_el(p_lista p, const int e){
    bool found = false;     // booleano per segnare quando trovo il nodo
    while ((p != NULL) && (!found))     // finche p diverso da NULL e
    {                                   // non trovo il nodo
        if (p->val == e) found = true;  // se lo trovo
        else p = p->next;               // altrimenti
    }
    return(p);                          // output
}
```

7.6.6 Stampare gli elementi di una lista

Utilizzo un iteratore per girare tutta la lista finché non arrivo al nodo finale, ovvero quello che punta a `NULL`

L'iteratore viene utilizzato per non cambiare l'informazione di `p` dato che viene passato per riferimento.

```
void stampa_el(p_lista p){
    p_lista iter = p;
    while (iter != NULL){
        cout<<iter->val <<" ";
        iter = iter->next;
    }
}
```

7.6.7 Inserimento in coda

Ancora qua il prof. sviluppa la funzione vedendo e correggendo il codice quando "scopre" determinati errori, riporto la versione finale della funzione.

```
p_lista tail_insert(p_lista head, int el){
    p_lista p = head;       // creo un nuovo puntatore
    if (head == NULL){      // Se mi viene riportata una lista vuota aggiungo
        head = new lista;   // direttamente a head
        head->val = 10;
        head->next = NULL;
    }else{                  // altrimenti ciclo finche non trovo la psz
        while (p->next != NULL){
            p = p->next;
        }
        p->next = new lista;// e poi creo un nuovo nodo
        p = p->next;        // e faccio la stessa cosa
        p->val = 10;
        p->next = NULL;     // essendo la coda next punta a NULL
    }
}
```

7.6.8 Rimozione dalla coda

Per la rimozione dalla coda dobbiamo iterare su `p->next->next`
Possiamo utilizzare anche due puntatori `p` e `p_after`
`p_after` punta al nodo successivo a `p`

doppiamo separare i casi di lista con un solo elemento da quelli di lista con almeno due elementi.

```
void tail_remove(p_lista head){
    p_lista p, p_after;
    if (head->next == NULL){
        delete head;
        head = NULL;
    }else{
        p = head;
        p_after = p->next;
        while (p_after->next != NULL){
            p = p_after;
            p_after = p_after->next;
        }
        delete p_after;
        p->next = NULL
    }
}
```

7.6.9 Esercizi sulle liste:

1. scrivere una funzione che prende in input una lista e un intero `n` e rimuove dalla lista tutti gli interi multipli di `n`, quindi ritorna la lista
2. implementare il **crivello di Eratostene**
3. scrivere un programma che consente all'utente di lavorare su una lista di interi con le seguenti operazioni:
   - aggiungere un elemento in fondo
   - vedere le informazioni dell'elemento corrente
   - andare avanti di uno
   - andare indietro di uno
   - eliminare l'elemento corrente
4. implementare il tipo di dato *pila* con le operazioni di:
   - creazione di pila vuota
   - `push` = inserimento di un elemento nella pila
   - `pop` = eliminazione di un elemento dalla pila
   - `top` = valore dell'elemento in testa alla pila
5. implementare il tipo di dato *coda* con le operazioni di:
   - creazione della coda vuota
   - `enqueue` = inserimento di un elemento nella coda
   - `dequeue` = eliminazione di un elemento dalla testa della coda
   - `top_el` = valore dell'elemento in testa alla coda
6. implementare il tipo di dato *insieme* con le operazioni di:
   - creazione dell'insieme voto
   - `is_in` = appartenenza di un elemento all'insieme
   - `union` = unione di due insiemi
   - `intersection` = intersezione di due insiemi

7.7 Liste bidirezionali

Alcune delle operazione base sulle liste (quelle degli esercizi) sono *scomode* da implementare su liste semplici, possiamo modificare leggermente la struttura dati `lista` per risolvere problemi come:

- accesso all'ultimo elemento
  1. nelle liste semplici è necessario scorrere tutta la lista
  2. è sufficiente tenere in una variabile il puntatore all'ultimo elemento della lista
- spostamento all'indietro
  1. nelle liste semplici è necessario scorrere tutta la lista cercando l'elemento che punta a quello corrente
  2. è sufficiente tenere in ogni elemento un puntatore al precedente
  3. nel primo elemento questo puntatore avrà valore `NULL`
  4. queste liste si chiamano **bidirezionali** o **con doppi puntatori**

```
1 struct blista{
2     int val;
3     blista *prec;
4     blista *next;
5 };
6
7 typedef blista* ptr_blista;
```

7.7.1 Esercizi sulle liste bidirezionali

1. prendere in input caratteri e creare una lista bidirezionale che li contenga.

   Stampare i caratteri inseriti in ordine inverso e in ordine normale

2. scrivere un programma che consente all'utente di lavorare su una lista bidirezionale di interi con le seguenti operazioni
   - aggiungere un elemento in fondo
   - vedere le informazioni dell'elemento corrente
   - andare avanti di uno
   - andare indietro di uno
   - eliminare l'elemento corrente

Chapter 8

Funzioni ricorsive

Prendiamo in considerazione il seguente codice:

Rimuove i multipli di `n` da una lista `p`

```
ptr_lista multiple_remove(ptr_lista p, int n) {
    ptr_lista p_init = p;
    ptr_lista p_old = p;
    while (p != NULL) {
        if ((p_init == p) && ((p->val) % n == 0)){
            p_init = p->next ;
            delete p ;
            p = p_init ;
            p_old = p ;
        } else if ((p_init != p) && ((p->val) % n== 0)){
            p_old->next = p->next ;
            delete p ;
            p = p_old->next ;
        } else {
            p_old = p ;
            p = p->next ;
        }
    }
    return(p_init);
}
```

La sua version ricorsiva, invece:

```
ptr_lista multiple_remove_ric(ptr_lista p, int n) {
    if (p == NULL) return(NULL) ;
    else if ((p->val) % n == 0) return (multiple_remove_ric(p->next, n)) ;
    else { p->next = multiple_remove_ric(p->next, n) ;
        return(p) ;
    }
}
```

È facile notare che la dimensione del codice di due funzioni che eseguono la stessa funzione è molto differente tra la versione **iterativa** e la versione **ricorsiva**

8.1 Divide et Impera

Una tecnica per risolvere i problemi è il

Divide et Impera

1. si scompone il problema originale in sottoproblemi (**divide**)
2. si risolvono i sottoproblemi
3. dalle soluzioni dei sottoproblemi si deriva la soluzione del problema orignale (**impera**)

8.2 Ricorsione

In molti casi quando si **divide** un problema in sottoproblemi, i sottoproblemi sono versioni o casi più semplici del problema generale

Una funzione è ricorsiva quando nel suo corpo c'è un'invocazione a sé stessa

Prendiamo ora un esempio da analizzare:

Vogliamo scrivere una funzione

`void write_vertical(int n)`

che prende un intero e lo stampa a video con tutte le cifre incolonnate partendo da quella più significativa

Algoritmo:

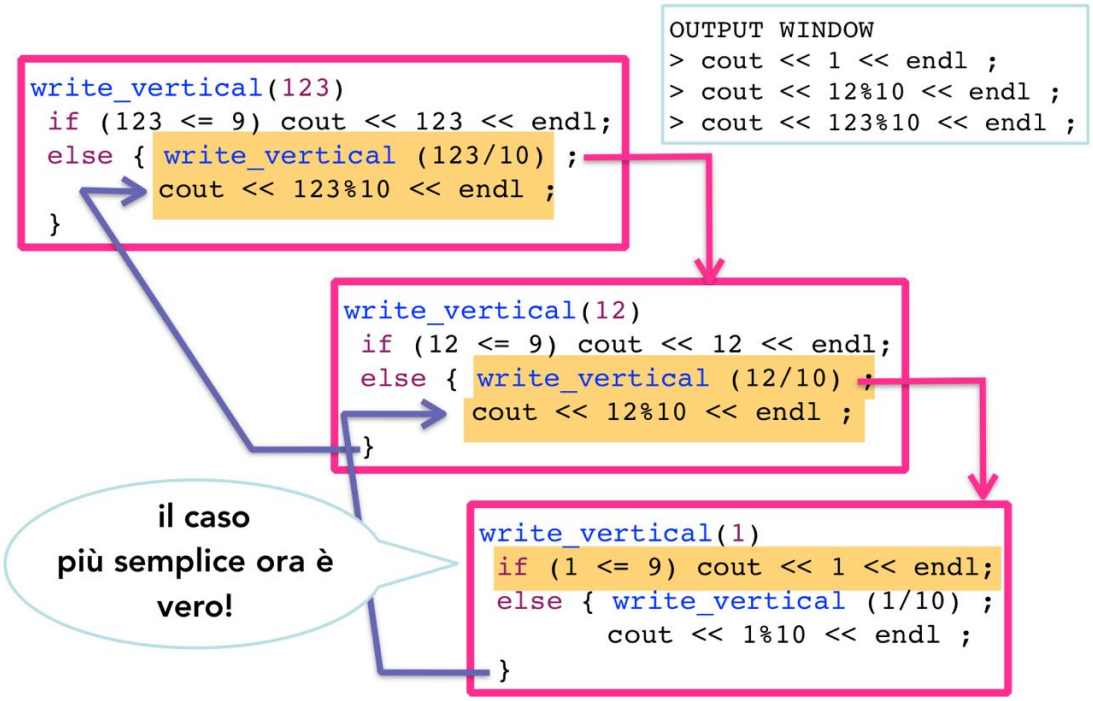
- Caso base: Se `n` è una sola cifra, allora scrivi il numero
- caso tipico:
  1. stampa il numero incolonnato meno l'ultima cifra
  2. scrivi l'ultima cifra

Osserviamo che il sottoproblema 1. è una versione più semplice del problema originale; il sottproblema 2. è il caso più semplice

```
1  void write_vertical(int n);
2      // Precondition: n >= 0
3      // Postcondition: stampa le cifre di n incolonnate a partire
4      // dalla piu significativa
5
6  void write_vertical(int n){
7      if (n <= 9) cout << n << endl;
8      else {
9          write_verticale(n/10);
10         cout << n%10 << endl;
11     }
12 }
```

esempi di output:

```
write_vertical(123)   1
                      2
                      3
```



Le chiamate ricorsive in `write_vertical` vengono eseguite

1. fermando temporaneamente l'esecuzione del chiamante in corrispondenza della chiamata ricorsiva
2. salvando le informazioni del chiamante per continuare l'esecuzione in seguito
3. valutando la chiamata ricorsiva
4. ripristinando l'esecuzione del chiamante

Quando raggiungiamo il caso base la ricorsione si ferma e le chiamate ritornano al livello superiore finché la prima chiamata di funzione (in ordine di tempo) non termina.

8.3 La ricorsione e le Pile

I computer, per eseguire una funzione ricorsiva, utilizzano le pile.

Una pila è una struttura dati di memoria simile a una "*pila di carta*"

- per mettere le informazioni sulla pila, si scrive su di un nuovo foglio di carta e lo si mette sulla cima della pila
- per recuperare le informazioni è possibile leggerle solamente dal foglio di carta che è sulla pila, che occorre buttare prima di leggere quelle del foglio sottostante

8.3.1 LIFO

La struttura della pila è detta `LIFO` o **last-in/first-out**, l'ultimo elemento che è stato inserito è il primo ad essere rimosso

Quando una funzione viene invocata il computer **"utilizza un nuovo foglio di carta"** Quando **l'invocazione** di una funzione esegue una chiamata ricorsiva:

1. l'esecuzione del chiamante viene messa in pausa (congelata) e le informazioni vengono salvate sul foglio di carta corrente (per consentire il ripristino dell'esecuzione più tardi
2. il foglio di carta corrente viene posto sulla pila
3. un **nuovo** foglio di carta è usato, sopra ci si copia la definizione e gli argomenti della funzione
4. l'esecuzione della funzione chiamata inizia

Invece quando **una chiamata ricorsiva termina**:

1. il computer "butta" via il foglio relativo alla funzione e recupera il foglio che si trova in testa alla pila
2. continua l'esecuzione corrispondente grazie alle informazioni memorizzate nel foglio
3. quando l'esecuzione termina, si scarta il foglio e si prende quello successivo
4. il processo continua finché non rimane alcun foglio

8.3.2 Record di attivazione

Il computer non usa veri e propri fogli di carta ma usa porzioni di memoria (`RAM`) dette: **record di attivazione**

- i record di attivazione **non contengono copie del codice** ma soltanto l'indirizzo della istruzione nell'unica copia del codice in memoria

Stack overflow

poiché ogni chiamata corrisponde a porre un nuovo record di attivazione in memoria, la **ricorsione infinita causa il consumo dello spazio riservato alla pila**

- la computazione termina con errore `out of memory`

8.4 Esercizi

1. scrivere una funzione ricorsiva che prende `n` e stampa `n` asterischi
2. scrivere `write_vertical` in maniera che le cifre vengano stampate dalla meno significativa alla più significativa
3. scrivere una funzione ricorsiva che prende `n` e stampa un numero di asterischi uguale alla somma dei quadrati dei primi `n` numeri naturali

Chapter 9

Classi

9.1 La programmazione object-oriented e le classi

9.1.1 Che cos'è una classe?

Una classe è un tipo di dato che specifica come i sui elementi, detti oggetti, possono essere creati e usati.

Per definire una classe occorre:

- definire i **campi** degli oggetti (tipo e identificatore)
- definire le funzioni della classe, dette **metodi**

Una classe **estende il concetto di struttura**.

le classi racchiudono **in un unico costrutto sintattico** la definizione dei valori e quella delle **operazioni** su quei valori.

Nelle strutture la connessione con le operazioni non c'è.

9.1.2 Utilità delle classi

in C++ le classi sono il mattone elementare per costruire grossi programmi. Sono anche molto utili per piccoli programmi

9.2 Campi e metodi

9.2.1 Un primo esempio di classe

definiamo ora una classe rectangle

- che ha **due campi** che memorizzano la base e l'altezza
- ed ha **due metodi**
  - **area** restituisce l'area del rettangolo relativo.
  - **perimeter** ritorna il perimetro del rettangolo relativo

```
1  class rectangle{
2      public:
3          int base;                                  // primo campo
4          int altezza;                               // secondo campo
5          int area(){                                // metodo area
6              return(abs(base*altezza));
7          };
8          int perimeter(){                           // metodo perimetro
9              return(2*abs(base) + 2*abs(altezza));
10         };
11 };
```

Con questa definizione della classe rectangle, nel main possiamo scrivere

```
int main(){
    rectangle x, y;     // come la definizione di una variabile

    x.base = 2;
    y.base = 3;
    x.altezza = -3;
    y.altezza = 4;
    cout<<x.area() <<" " <<y.perimeter;
    return 0;
}
```

E l'output nel terminale sarà:

```
-6 14
```

Perché viene calcolata l'area del rettangolo x e il perimetro del rettangolo y (sono due rettangoli distinti).

- Gli oggetti vengono dichiarati usando il nome della classe come tipo.
- NOTA: l'accesso ai campi (gli identificatori della classe base e altezza) funziona come per le strutture, si chiama l'oggetto "x" e si aggiunge un punto "." per l'identificatore desiderato.
- L'accesso ai metodi è, invece, simile a quello dei campi.

Problema:
con questa implementazione di rectangle l'utente ha accesso ai campi della classe (base e altezza) **non è necessario che possa farlo**.
E **se si decide di cambiare** l'implementazione (il nome dei campi o il loro tipo) sarà necessario cambiare tutto il codice.

9.2.2 Dichiarazioni delle classi

In C++ è anche possibile definire i metodi all'esterno della classe attraverso la notazione "::"

```
class rectangle{
    public:
        int base;
        int altezza;
        int area();
        int perimeter();
};
    ...
int rectangle::area(){
    return(base*altezza);
};
int rectangle::perimeter(){
    return(2*base + 2*altezza);
};
```

Problemi: così facendo perdiamo i vantaggi del concetto di classe, ovvero avere un unico costrutto o un'unica parte di codice per finalizzata ad un determinato scopo.

9.2.3 Primi esercizi

1. definire una classe cerchio (ispirandosi alla classe rettangolo) con i metodi per area e circonferenza.
    - dopo aver definito la classe, scrivere un programma che crea un cerchio e da in output sul terminale l'area.
2. nel definire la classe cerchio avete dovuto scegliere se utilizzare un campo raggio o un campo diametro;
    - modificare la classe cerchio precedentemente definita cambiando questa scelta (usare il raggio al posto del diametro o il diametro al posto del raggio).
    - il programma precedente funziona anche con la nuova classe cerchio? O bisogna cambiare anche altre parti del programma?

9.3 Encapsulation

9.3.1 Incapsulamento

Nell'esercizio precedente (2) effettuare un cambio alla struttura interna del cerchio **richiede** un cambio del programma che usa il cerchio
Questo non va bene perché:

- rende il programma difficile da modificare;
- non posso far sviluppare la classe e il codice che lo usa a persone diverse.;
- va contro a ciò che la programmazione object-oriented cerca di fare per semplificare il lavoro.

Questo succede perché non abbiamo separato la **parte pubblica** e la **parte privata**.

9.3.2 Membri private e public

La definizione

```
class nome_classe{
    protected:
        campo_1;
        campo_2;
    public:
        metodo_1(){
            ...
        };
        metodo_2(){
            ...
        };
};
```

Ci da la possibilità di "Proteggere" i campi della classe. Così facendo non sarà più possibile accedere direttamente ai campi come facevamo prima;

```
        ...
    x.altezza = 4;
    x.base = 6;
        ...
```

Ma potremmo accedere solamente a ciò che è presente nella parte `public` (Aperta al pubblico, dove il pubblico è tutto ciò che sta al di fuori della classe)

Essendo i campi (come, nell'esempio della classe rettangolo, base e altezza) non accessibili abbiamo bisogno di un metodo nella parte `public` che ci possa far modificare i campi sotto protected. Vedremo più avanti un metodo *specifico* (make) per questo lavoro di modifica dei campi protetti.

Una volta chiarito questo possiamo andare a riformulare la classe rettangolo:

```
class better_rectangle{
    protected:
        int base;
        int altezza;
    public:
        int area(){
            return(base*altezza);
        };
        int perimeter(){
            return(2*abs(base) + 2*abs(altezza));
        };
        void make(int b, int n){
            base = abs(b);
            altezza = abs(a);
        }
};
```

9.3.3 Incapsulamento - Riepilogo

Un oggetto contiene una parte `pubblica` e una parte `privata`

- la parte **Pubblica** è visibile da chi usa l'oggetto
- la parte **Privata** non lo è
  - si dice che la parte privata è *incapsulata*

Se un programmatore modifica la parte privata lasciando inalterata quella pubblica, chi lavora sullo stesso progetto (ma logicamente non su quella stessa classe) non se ne accorge perché non gli è accessibile.

9.3.4 Cosa definire Public e cosa Protected

Solitamente vengono posti sotto `protected` tutte le informazioni legate all'implementazione, quindi:

- tutti i campi;
- tutti i metodi ausiliari;

9.4 costruttori

Quando creiamo l'oggetto di una classe bisogna inizializzare i campi. Conviene quindi creare un metodo `public` che li inizializza, questo metodo è **necessario** per i campi protected poiché da fuori la classe non è possibile accedere a ciò che sta sotto protected.

C++ da la possibilità di definire un metodo il cui identificatore (il tipo della funzione) è lo stesso della classe;
Questo metodo è detto **COSTRUTTORE**

```
class best_rectangle{
    protected:
        int base;
        int altezza;
    public:
        int area(){
            return(base*altezza);
        };
        int perimeter(){
            return(2*base + 2*altezza);
        };
        best_rectangle(int a, int b){
            base = abs(b);
            altezza = abs(a);
        };
};
```

I costruttori non hanno tipo di ritorno (è implicitamente la classe corrispondente). Questo perché il loro lavoro è quello di inizializzare i campi protetti per far si che siano disponibili agli altri metodi.

Ora l'inizializzazione dei campi della funzione avviene con l'inizializzazione di ogni oggetto:

```
int main(){
    best_rectangle x(5, 3);                    // sono equivalenti
    best_rectangle x = best_rectangle(5, 3);   // sono equivalenti
    return 0;
}
```

I Costruttori **non si invocano come gli altri metodi**, non è quindi possibile scrivere:

```
    best_rectangle x;
    x.best_rectangle(5, 3);
    // error: no matching function for call
    // to 'best_rectangle::best_rectangle()'
```

Inoltre è possibile definire **i valori di default** nel costruttore. Se nella dichiarazione dell'oggetto il costruttore non viene invocato allora i campi sono inizializzati con i valori di default.

```
    best_rectangle(int b = 0, int a = 0){
        base = abs(b);
        altezza = abs(a);
    };
```

9.4.1 Esercizi sui costruttori

1. Definire una classe `frazione` con un opportuno costruttore ed i metodi `stampa`, `moltiplica`, e `inverso`.
   - il metodo `moltiplica` prende come parametri due frazioni e mette il risultato nella frazione su cui è invocato.
   - il metodo `inverso` inverte la frazione su cui è invocato.

9.5 Puntatori a oggetti

È possibile definire **puntatori a oggetti**, per accedere agli elementi (campi e metodi `public`) si utilizzano gli stessi meccanismi delle strutture.

```
int main(){
    best_rectangle a(3, 5);
    best_rectangle *b, *c;
    best_rectangle *d = new best_rectangle(2, 3);
    b = new best_rectangle();
    c = &a;
    cout<< a.area();
    cout<< b->area();
    delete b; delete c;
    return 0;
}
```

9.6 Tipi di dati astratti

9.6.1 Stack

```
struct pila{
    int val;
    pila *next;
};

typedef pila *p_pila;

class stack{
    protected:
        p_pila elem;
    public:
        stack(){
            elem = NULL;
        };
        void push(int e){
            p_pila x = new(pila);
            x->val = e;
            x->next = elem;
            elem = x;
        };
        void pop(){
            p_pila x;
            if(elem != NULL){
                x = elem;
                elem = elem->next;
                delete(x);
            }
        };
        int top(){
            if(elem != NULL){
                return(elem->val);
            }
        };
};
```

La prima implementazione ha come unico campo un nodo di una struttura che viene inizializzata sempre vuota dal costruttore, quindi quando dichiariamo un oggetto di questo tipo dovremo andare a chiamare le funzioni successive per poter aggiungere (`push`) o rimuovere (`pop`) elementi dallo stack.

```
// definizione alternativa
// utilizza puntatori a stack
class stack{
    typedef stack *p_stack;
    protected:
        int val;
        p_stack next;
    public:
        stack(){
            val = 0;
            next = NULL;
        };
        bool is_empty(){
            return(next == NULL);
        };
        void push(int e){
            p_stack tmp = new stack();
            tmp->val = e;
            tmp->next = next;
            next = tmp;
        };
        void pop(){
            if(next != NULL){
                p_stack tmp = next;
                next = next->next;
                delete(tmp);
            }
        };
        int top(){
            if(next != NULL){
                return(next->val);
            }
        };
}
```

La seconda implementazione prevede che gli identificatori che utilizzavamo nella struttura dati prima diventino i campi della classe. NON possiamo però usare stack *next, perché le strutture dati non possono essere ricorsive. Per il resto funziona come prima con le classiche operazionei `push`, `pop`, `top`, etc.

9.6.2 Stack con array

Vediamo ora un'implementazione dello stack ma utilizzando un array.

```
const int lenght = 1000;
class stackWithArray{
    protected:
        int pila[lenght];
        int num_el;
    public:
        stackWithArray(){
            num_el = 0;
        };
        bool is_empty(){
            return(num_el == 0);
        };
        void push(int e){
            if(num_el == lenght) cout<<"OUT OF MEMORY\n";
            else {
                pila[num_el] = e;
                num_el = num_el + 1;
            }
        };
        void pop(){
            if (num_el != 0) num_el = num_el - 1;
        };
        void top(){
            if (num_el != 0) return(pila[num_el - 1]);
        };
};
```

9.7 this

Esiste una variabile predefinita "`this`" che abbiamo a disposizione quando scriviamo il codice di un metodo, `this` contiene un puntatore all'oggetto corrente quindi ogni campo può essere referenziato con uno dei due modi equivalenti

```
    elem = 10;
    this->elem = 10;
```

`this` è utile quando si deve passare l'oggetto corrente come argomento di una funzione o di un metodo; oppure quando si deve tornare come valore un puntatore all'oggetto corrente

---

9.8 Esercizi Finali

- implementare il tipo di dato coda con le classi non facendo nessuna assunzione sulla dimensione delle code
- implementare il tipo di dato insieme di interi con le classi

Chapter 10

Ereditarietà

Spesso non si vuole definire una classe a partire dal nulla ma vogliamo **definirela a partire da un'altra classe**. Per esempio uno `studente` è anche una `persona` e se abbiamo già definito una classe `persona` vorremmo riusare il codice già scritto.

Uno studente è quindi una **sottoclasse** di `persona` e:

- ha **tutti i campi** di persona
- ha tutti i metodi di persona, eventualmente modificati (tranne il costruttore)
- può avere altri metodi e altri campi propri

Infatti come `persona` uno `studente` ha `nome`, `cognome`, `indirizzo`, `telefono`, ... E in particolare, uno studente ha una lista di esami, una media, ... campi che una `persona` generica non ha.

Ma quindi qual è il senso dell'ereditarietà?

- non si riscrivono tutti i campi e i metodi di una classe
- si scrivono solamente quelli nuovi e/o modificati
- riutilizzare il codice aiuta a risparmiare
  - tempo di scrittura
  - tempo di debugging
  - tempo di modifica

10.1 Terminologia e Sintassi

Partiamo con un esempio e definiamo i termini da questo esempio:

```
1  class persona{
2      protected:
3          char nome[50];
4          int eta;
5      public:
6          persona(char n[]="", int e = 0){
7              strcpy(nome, n);
8              eta = e;
9          };
10         void presentati(){
11             cout<<"Sono " <<nome <<" e ho "<<eta <<"anni \n";
12         }
13 };
14
15 class studente: public persona{
16     protected:
17         double media;
18     public:
19         studente(char n[], int e, double m){
20             strcpy(nome, n);
21             eta = e;
22             media = m;
23         };
24         void presentati(){
25             cout<<"Sono " <<nome <<", ho " <<eta <<" anni e "
26             <<media <<" di media.";
27         };
28 };
```

10.1.1 Terminologia:

- `studente` si dice classe derivata, o **sottoclasse**, o classe figlio di `persona`
- `persona` si dice classe base, o **superclasse**, o classe padre di `studente`
- la modifica di un metodo della classe padre nella sottoclasse (ln `24-27`) si chiama **overriding**

10.1.2 Sintassi:

Alla linea `15` andiamo a definire che `studente` è una sottoclasse di `persona` definendogli il livello di accesso alla classe padre

```
1 class nomeFiglio: public nomePadre{
2     // campi nuovi e metodi nuovi o modificati
3 };
```

- `nomeFiglio` è il nome della nuova classe, `nomePadre` il nome della classe da cui si eredita
- la nuova classe **ha tutti i campi della classe vecchia**, più quelli nuovi
- la nuova classe **ha tutti i metodi della classe vecchia** (tranne il costruttore), eventualmente modificati, più quelli nuovi
- il prototipo del metodo modificato deve essere uguale a quello del corrispondente metodo nel padre (`presentati()` nell'esempio)

10.1.3 Ereditarietà dei Costruttori

*I costruttori **NON** vengono ereditati*

Prima di invocare il costruttore di una sottoclasse viene invocato il costruttore senza parametri della superclasse e se il costruttore della superclasse non esiste si ha un `errore`; in questo modo il costruttore della sottoclasse può assumere che i campi della superclasse siano già stati inizializzati.

10.2 Sottotipaggio

una sottoclasse è un SOTTOTIPO della superclasse

È quindi possibile utilizzare un oggetto della sottoclasse in qualunque contesto serva un oggetto della superclasse.

- un oggetto della sottoclasse **può essere assegnato** ad una variabile della superclasse (ln `35`).
- un oggetto ella sottoclasse **può essere passato come parametro** ad una funzione che si aspetta un oggetto della superclasse.
- un oggetto della sottoclasse **può essere ritornato come valore di ritorno** se il tipo di tale valore deve essere una superclasse.
- un oggetto della sottoclasse **può essere puntato** da un puntatore alla superclasse

Il contrario di queste 4 affermazioni (ovvero da superclasse a sottoclasse) non è vero!

```
class rectangle{
    protected:
        double base;
        double altezza;
    public:
        rectangle(int b = 0, int a = 0){
            base = abs(b);
            altezza = abs(a);
        };
        double area(){
            return(base * altezza);
        };
        double perimeter(){
            return(2*base + 2*altezza);
        };
};

class triangle_rect: public rectangle{
    public:
        triangle_rect(int b, int a){
            base = abs(b);
            altezza = abs(a);
        };
        double area(){
            return((base*altezza)/2);
        };
        double perimeter(){
            return(base + altezza + sqrt(base*base + altezza*altezza));
        };
};

int main(){
    triangle_rect t(3, 4);
    rectangle r(2, 3);
    r = t;
    cout<<r.perimeter() <<" " <<t.perimeter();
}
```

`Output: 14 12`

10.3 Costruttori e overriding

10.3.1 Riutilizzo dei costruttori

Una sottoclasse può dichiarare quale costruttore della classe base vuole utilizzare

```
class persona{
    ...
    persona(char n[], int e){
        strcpy(nome, n);
        eta = e;
    };
    ...
};
class studente{
    ...
    studente(char n[], int e, double m):persona(n, e){
        media = m;
    };
};
```

Scrivendo `:persona(n, e)` alla ln11 possiamo richiamare il costruttore della classe base per risparmiare tempo e righe di codice.

10.3.2 Overriding

La ridefinizione di un metodo nella sottoclasse si chiama **overriding** e ci da la possibilità di accedere al metodo omonimo nella superclasse
Possiamo fare lo stesso "giochino" con gli altri metodi, cambia solo la sintassi per richiamarli.

Sintassi: Lo si richiama con `superclasse::metodo`

```
class persona{
    ...
    void presentati(){
        cout<<"Sono " <<nome <<" e ho " <<eta << " anni\n";
    };
    ...
};
class studente{
    ...
    void presentati(){
        persona::presentati();
        cout<<"Ho una media voti " <<media <<"\n";
    };
    ...
};
```

10.4 Gerarchia di classi

Una sottoclasse può essere a sua volta la superclasse di un'altra classe.

- `studente` è una sottoclasse di `persona`, e una *superclasse* di `laureando`.
- un `laureando` ha i metodi e i campi di `persona`, quelli di `studente` più altri, ad esempio `relatore` e `titoloTesi`

Lavorando e pensando in questa maniera si possono definire gerarchie complicate.

10.5 Esercizi

1. Definire una classe `citta` con campi `nome` e `numeroAbitanti` con un opportuno costruttore e metodi `descrizione` (che stampale informazioni sulla città) e `camiaAbitanti`, che cambia il numero di abitanti.
2. Definire una classe `capoluogo` che è una `citta` con in più l'informazione sulla `regione` di cui è capoluogo.
3. Definitr una classe `capitale` che è un `capoluogo` con in più l'informazione sulla `nazione` di cui è capitale.
4. Creare gli oggetti per `Cesenatico`, `Bologna`, `Torino` e `Roma` e stamparne le informazioni.