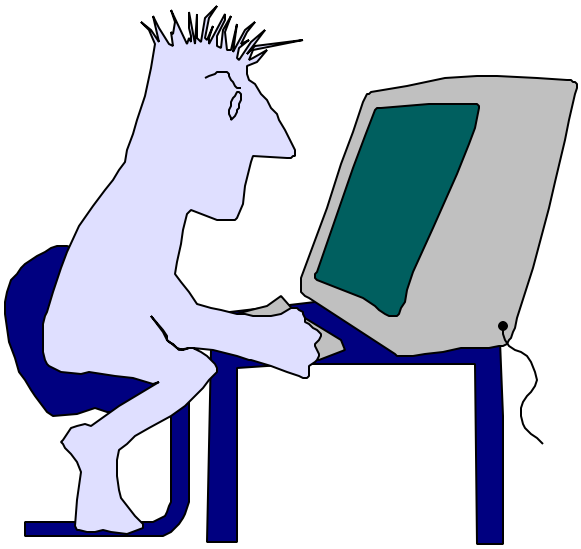
Linguaggio di programmazione

C++



Autore: Roggero Piero

1° Modulo

- Introduzione
  - Struttura di un elaboratore
  - Assembler
  - Fasi di produzione
  - Struttura di un programma
- Lessico
  - Parole chiave
  - Identificatori
  - Variabili
  - Tipi
- Istruzioni
  - Di assegnamento
  - Composte
  - Condizionali
    - If
    - If-else
    - Switch-case
  - Iterative
    - While
    - Do-while
    - For
    - Break
    - Continue
- Vettori
- Costanti
- Operatori
  - Aritmetici
  - Autoincremento
  - Regole
    - Precedenza
    - Associatività
  - Virgola
  - Ternario
  - Relazionali
  - Logici
  - Conversione
- Input/Output
  - Printf
  - Scanf
- Leggibilità
- Debugger

Programmazione in C++

Un **programma** è una sequenza di istruzioni, eseguibili da un elaboratore, per il trattamento di informazioni relative ad un dato problema.

Programmare significa codificare in un dato linguaggio la soluzione ad un problema.

Caratteristiche di un buon programma:

- Corretto
- Facilmente mantenibile
  - Leggibile
  - Documentazione appropriata
- Portabile su più piattaforme
  - Indipendente dall'hardware
- Ottimizzato

La **leggibilità** di un programma rappresenta un fattore importante quando il software è di dimensioni notevoli e la realizzazione è affidata a più persone, inoltre è un fattore che facilita le modifiche e la ricerca degli errori.

Un **linguaggio di programmazione** è un insieme di parole chiave e di regole che definiscono un alfabeto e una grammatica opportuna a codificare un programma.

La **sintassi** è l'insieme di regole che stabiliscono le modalità di scrittura delle istruzioni.

La **semantica** specifica il significato associato alle regole sintattiche.

Caratteristiche del linguaggio C++

- E' un linguaggio strutturato, fornisce i costrutti fondamentali per il controllo del flusso di elaborazione.
- Adotta un controllo forte dei tipi delle variabili.
- E' un linguaggio relativamente a basso livello; non fornisce operazioni per trattare direttamente oggetti compositi come stringhe, insiemi, liste o vettori.
- E' un linguaggio a formato libero.
- E' un linguaggio minimale, dispone solo degli operatori essenziali, tutto il resto si ottiene tramite funzioni di libreria.

Vantaggi del linguaggio C++

Efficiente; permette di scrivere dei programmi compatti

Fornisce un controllo completo sulle operazioni.

Consente la massima libertà nella organizzazione del lavoro.

Struttura di un elaboratore

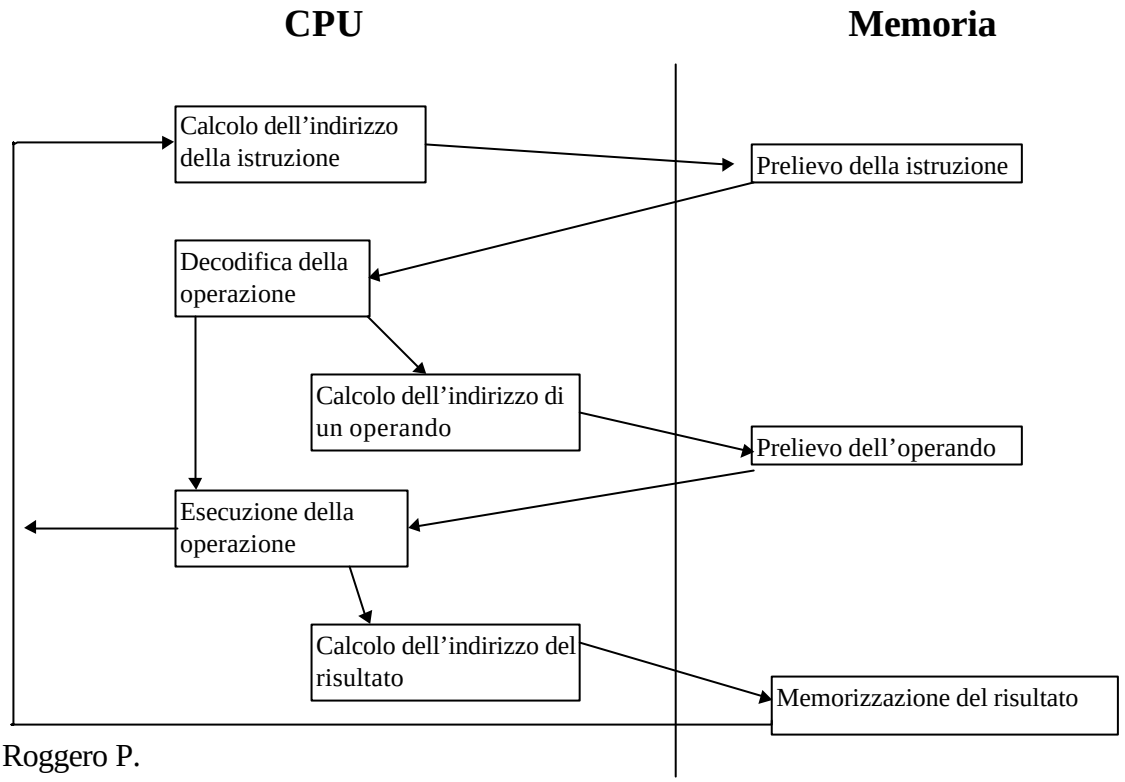
Le parti principali di un elaboratore sono:

CPU (Central Processing Unit)
La **memoria**
Le unità di **input** e **output**

La CPU è costituita da una unità di controllo che provvede alla esecuzione delle istruzioni che compongono un programma e da una unità aritmetico logica che esegue le operazioni di calcolo.
I **registri** sono unità di memoria veloci interne alla CPU , i principali sono:

Program Counter	contiene l'indirizzo della successiva istruzione da eseguire
Istruction Register	contiene l'istruzione in esecuzione
Data Registers	contengono gli operatori della istruzione in fase di elaborazione

La memoria centrale è il dispositivo su cui vengono caricati sia il programma che i dati su cui l'elaboratore opera.
Essa è composta da un certo numero di celle ciascuna delle quali può contenere una informazione binaria; ogni cella è individuata da un numero intero che rappresenta il suo indirizzo.
L'unità di misura della memoria è il **byte** che corrisponde a 8 bit, si utilizza come riferimento anche la **parola** che risulta un multiplo del byte (16 bit, 32 bit, 64bit).
Le unità di ingresso e di uscita rappresentano i dispositivi periferici (tastiera, video, disco, stampante) che consentono di comunicare con l'elaboratore per introdurre i dati e per ottenere i risultati.
L'istruzione macchina è l'operazione eseguibile direttamente dall'hardware dell'elaboratore, esse vengono codificate in forma binaria.
Linguaggio macchina è insieme delle istruzioni macchina.
Ogni elaboratore ha il proprio linguaggio macchina, quindi un algoritmo per essere eseguito deve essere codificato in linguaggio macchina.
Le istruzioni macchina che lo compongono devono essere allocate in memoria centrale in celle contigue secondo indirizzi crescenti, dopo di che l'indirizzo della cella che contiene la prima istruzione da eseguire viene posta nel registro PC.
L'elaborazione risulta un ciclo continuo secondo lo schema Von Neuman.



L'esempio seguente rappresenta il codice Assembler e binario della operazione :
c = a * b

Assembler	Speudo macchina	IND	Macchina
LODD A	LODD 98	90	0000 0000 0110 0010
PUSH	PUSH	91	1111 0100 0000 0000
LODD B	LODD 99	92	0000 0000 0110 0011
PUSH	PUSH	93	1111 0100 0000 0000
CALL PMUL	CALL 100	94	1110 0000 0110 0100
IR: STOD C	STOD 97	95	0001 0000 0110 0001
JUMP STOP	JUMP 128	96	0110 0000 1000 0000
C: INTEGER	C	97	
A: INTEGER	A	98	
B: INTEGER	B	99	
PMUL: DESP 2	DESP 2	100	1111 1110 0000 0010
LODL A	LODL 4	101	1000 0000 0000 0100
JNZE ANOTZ	JNZE 105	102	1101 0000 0110 1001
LOCO 0	LOCO 0	103	0111 0000 0000 0000
JUMP DONE	JUMP 126	104	0110 0000 0111 1110
ANOTZ: LODL B	LODL 3	105	1000 0000 0000 0011
JNZE BNOTZ	JNZE 109	106	1101 0000 0110 1101
LOCO 0	LOCO 0	107	0111 0000 0000 0000
JUMP DONE	JUMP 126	108	0110 0000 0111 1110
BNOTZ: LOCO 0	LOCO 0	109	0111 0000 0000 0000
STOL P	STOL 1	110	1001 0000 0000 0001
LOCO 1	LOCO 1	111	0111 0000 0000 0001
STOL J	STOL 0	112	1001 0000 0000 0000
LODL A	LODL 4	113	1000 0000 0000 0100
JNEG L2	JNEG 125	114	1100 0000 0111 1101
JZER L2	JZER 125	115	0101 0000 0111 1101
L1: LODL P	LODL 1	116	1000 0000 0000 0001
ADDL B	ADDL 3	117	1010 0000 0000 0011
STOL P	STOL 1	118	1001 0000 0000 0001
LOCO 1	LOCO 1	119	0111 0000 0000 0001
ADDL J	ADDL 0	120	1010 0000 0000 0000
STOL J	STOL 0	121	1001 0000 0000 0000
SUBL A	SUBL 4	122	1011 0000 0000 0100
JNEG L1	JNEG 116	123	1100 0000 0111 0100
JZER L1	JZER 116	124	0101 0000 0111 0100
L2: LODL P	LODL 1	125	1000 0000 0000 0001
DONE: INSP 2	INSP 2	126	1111 1100 0000 0010
RETN	RETN	127	1111 1000 0000 0000

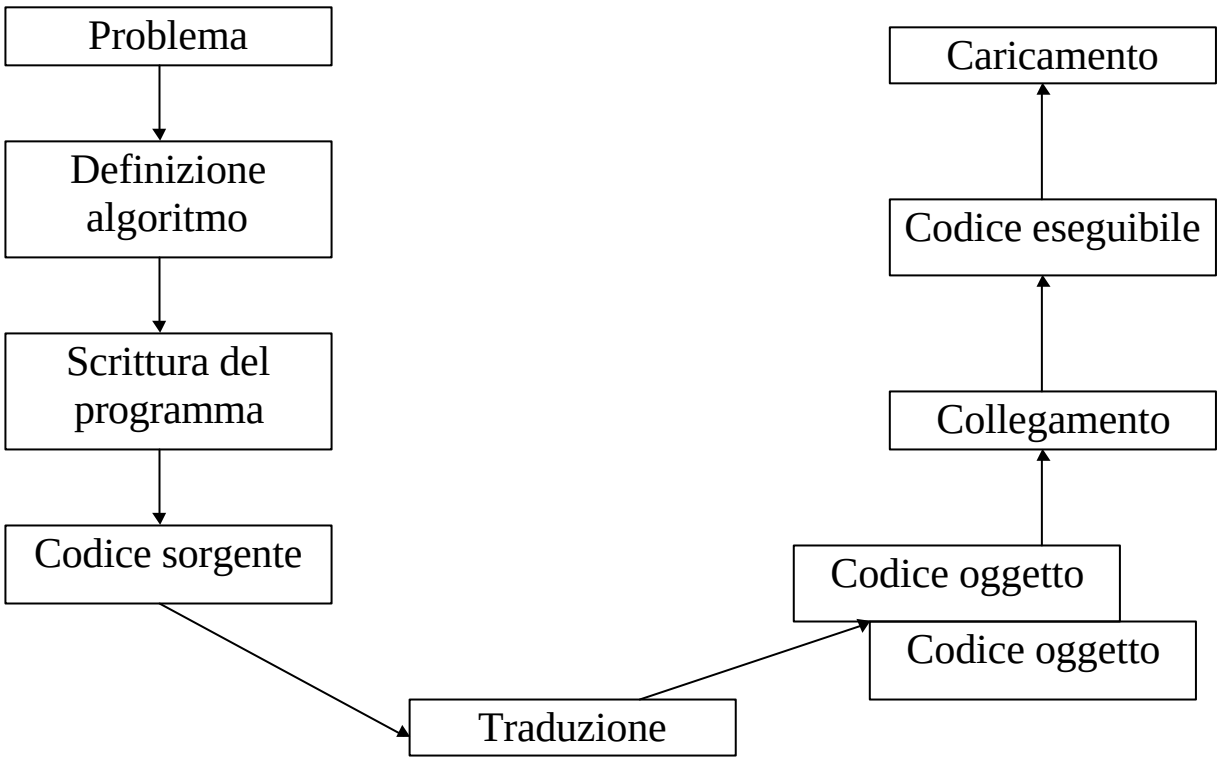
Il linguaggio assembler è un linguaggio mnemonico per rappresentare le istruzioni macchina.

Significato delle istruzioni assembler del codice precedente:

LODD	Vai all'indirizzo x e metti in accumulatore
LODL	Carica nell'accumulatore un elemento dello stack
LOCO	Carica nell'accumulatore una costante
PUSH	Carica sullo stack
DESP	Decrementa lo stack pointer
INSP	Incrementa lo stack pointer
CALL	Chiama la subroutine
STOD	Memorizza il valore dell'accumulatore nella locazione indicata
STOL	Memorizza il valore dell'accumulatore nello stack(n)
JUMP	Salto incondizionato
JNZE	Salto se non zero
JNEG	Salto se negativo
JZER	Salto se zero
ADDL	Addiziona locazione acc + stack(n)
SUBL	Sottrai locazione acc - stack(n)
RETN	Return

Un **linguaggio di programmazione** ad alto livello consente invece di descrivere un algoritmo secondo un formalismo più vicino al linguaggio umano ma necessita di una fase di traduzione in linguaggio macchina che si può realizzare con un compilatore.

Fasi di produzione



Struttura di un programma

Un programma è una sequenza di **token** separati da spazi.
Un **token** è un stringa di caratteri alfanumerici che determinano un oggetto del linguaggio.

I token possono essere:

Parole chiave	parole facenti parte del linguaggio
Identificatori	parole inventate dal programmatore per indicare degli oggetti, costanti, variabili funzioni.
Costanti	valori numerici e letterali.
Stringhe	sequenza di caratteri racchiuse da doppi apici
Operatori	simboli per indicare delle operazioni aritmetiche, logiche, di confronto
Altri separatori	simboli del linguaggio che determinano la sintassi del programma.

Ogni programma C++ deve presentare una funzione main(), e il programma inizia con l'attivazione di questa funzione.

Esempio di programma minimo in C:
void main() { }
Di norma un programma genera dell'output

Esempio:
```
#include <iostream.h>                    // Libreria standard di input ed output
void main()
{
        cout << "Hello, World\n";        // "\n" carattere di ritorno a capo
}
```

Esempio di commento stile C
```
/* Commento che prosegue
 * su più righe
 */
```

Esempio di commento stile C++
// Commento fino a fine riga
I commenti C++ non sono compatibili con il C
Esempio:
```
int a = b //* commento */ c;
```

Esempio di programma

```
/* scontr.cpp
 * Calcola lo scontrino di un acquisto
 */

#include <iostream.h>

float Tassa(float);

void main()
{
        float acquisto, amm_tassa, totale;
        cout << "\nAmmontare dell'acquisto:  ";
        cin  >> acquisto;
        amm_tassa = Tassa(acquisto);
        totale = acquisto + amm_tassa;
        cout << "\nSpesa      =  " << acquisto;
        cout <<"\nTassa      =  " << amm_tassa;
        cout << "\nTotale     =  " << totale;
}

float Tassa(float spesa)
{
        float percentuale = 0.065;
        return (spesa * percentuale);
}
```

Parole chiave:	include, float, void, main, return
Identificatori	Tassa, acquisto, amm_tassa, totale, spesa, percentuale
Costanti	0.065
Stringhe	“\nAmmontare dell’acquisto”, “%f”, “\nSpesa = %f”
Operatori	= + *
Altri separatori	{ } ( ) ; ,

Parole chiave

Vengono dette **parole chiave o riservate** gli identificatori ai quali il linguaggio attribuisce un ben preciso significato e che quindi possono essere utilizzate solo in determinati contesti per lo scopo loro assegnato. Pertanto, non possono essere usate per denotare variabili o funzioni.

Elenco:

- Denominatori di tipo elementare
  char, double, float, int, void

- Modificatori e qualificatori di tipo
  const, long, short, signed, unsigned, volatile

- Denominatori di classe di memoria
  auto, extern, register, static

- Costruttori di tipo complessi
  enum, struct, typedef, union, class

- Gli operatori
  defined, sizeof, delete, new

- Costrutti operativi
  break, case, continue, default, do, else, for, goto, if, return. switch, while
  asm, catch, friend, inline, operator, private, protected, public, template, this, throw, try, virtual

Identificatori

Un identificatore è una sequenza arbitrariamente lunga di lettere e cifre.

- Devono iniziare con un carattere alfabetico od '_'

- Fa differenza tra caratteri minuscoli e maiuscoli

- Non devono essere uguali ad una parola chiave del linguaggio.

- Nello stesso ambiente i nomi devono essere diversi.

- In generale solo i primi 31 caratteri sono significativi.

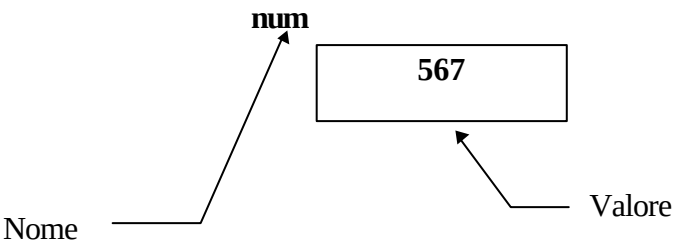
Ogni identificatore e ogni espressione possiede un tipo che determina le operazioni eseguibili.
Una dichiarazione è una istruzione che inserisce un nome nel programma e specifica un tipo per questo nome.
Esempio:

```
int inch;
```

Una dichiarazione precisa la visibilità di un nome, che può essere usato all'interno di una specifica parte del programma.

Variabile

area di memoria identificata da un nome che può contenere un valore di un certo tipo.



In C le variabili vanno sempre dichiarate prima di essere riferite.

Caratteristiche di una variabile:

- nome o identificatore
denominazione usata per poterla utilizzare in un certo ambiente.
- tipo
caratteristiche del contenuto.
- classe
modalità di utilizzo
- valore
contenuto della variabile in un dato momento della esecuzione del programma
- scope
ambiente in cui la variabile è visibile
- indirizzo
posizione di memoria in cui viene allocata la variabile

Dobbiamo distinguere i termini:

dichiarazione
rende note le proprietà di una variabile (nome, tipo)
definizione
provoca anche l'allocazione di un'area di memoria riservata a quella variabile.

Sintassi della dichiarazione di una variabile

[classe] tipo nome [= valore-iniziale];

Esempi di dichiarazioni:

```
int num;
float rad = 50.2;
static int cont = 0;
```

Esempi di utilizzo:

```
num = 45;
rad = num * rad;                    // = operatore di assegnamento
```

Ritorna

```
/* Oggetto.cpp
 * Trasforma una frazione in un numero decimale
 */

#pragma hdrstop
#include <condefs.h>
#include <iostream.h>

//---------------------------------------------------------------------------
#pragma argsused
void Attesa(void);
int main(int argc, char* argv[]) {
   float num, denom;
   float valore;

   cout << "Converte una frazione in un numero decimale" << endl
        << "Numeratore:   ";
   cin  >> num;
   cout << "Denominatore: ";
   cin  >> denom;

   valore = num / denom;

   cout << endl << num << " / " << denom << " = " << valore << endl;

   Attesa();
   return 0;
}
void Attesa(void) {
  cout << "\n\n\tPremere return per terminare";
  cin.ignore(4, '\n');
  cin.get();
}
```

Tipi di dati fondamentali

Il tipo di una variabile è l'insieme dei valori che essa può contenere.

char Insieme dei caratteri ASCII

int Insieme dei numeri interi

float Insieme dei numeri floatting point in singola precisione

double Insieme dei numeri floatting point in doppia precisione

Per i tipi int si possono utilizzare i qualificatori:

short, long, signed, unsigned

Per il tipo double si può utilizzare il qualificatore

long

Fare riferimento all'header standard <limit.h> per la definizione dei valori minimi e massimi di ogni tipo.

Tipi derivati:

vettori Insieme di oggetti tutti dello stesso tipo
Esempio: int vett[50];

puntatori Variabili che contengono l'indirizzo ad oggetti di un certo tipo
Esempio: float *p = &c;

strutture Insieme composto da una sequenza di oggetti di tipo diverso

```
struct record {
        int a;
        float b;
        char *msg;
} info;
```

union Insieme in grado di contenere uno fra diversi oggetti di diverso tipo.

```
union rec {
        int b;
        double h;
} buf;
```

funzioni Procedure che restituiscono oggetti di un certo tipo
Esempio: float media(int a, int b)

Qualificatori di tipo

const Indica che l'oggetto associato non può essere modificato

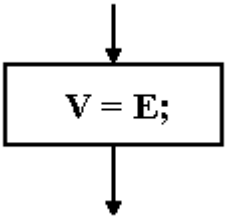
volatile Indica al compilatore di eliminare l'ottimizzazione che si potrebbe avere nella codifica del codice in modo tale che l'oggetto venga valutato in accordo con le regole di sequenza del C.

Esempio: Per una macchina con un input/output mappato in memoria, un puntatore ad un registro di device potrebbe essere qualificato volatile per impedire che il compilatore rimuova i riferimenti apparentemente ridondanti effettuati tramite questo puntatore (al riguardo non esistono semantiche indipendenti dall'implementazione).

Ritorna

```
/* Tipi.cpp
 * Uso dei tipi base
 */
#pragma hdrstop
#include <condefs.h>
#include <iostream.h>
#include <iomanip.h>
#include <ctype.h>    /* Libreria funzioni sui caratteri */
#include <math.h>     /* Libreria funzioni matematiche */

#define MAX 20

//---------------------------------------------------------------------------
#pragma argsused
int   intero, q, r;
float rad, seno;
char  car;
void Attesa(void);

int main(int argc, char* argv[]) {
/* Utilizzo del tipo int */
   cout << "Scrivere un numero intero\n";
   cin  >> intero;
   q    = intero / MAX;
   r    = intero % MAX;
   cout << setw(5) << intero << " ="
        << setw(5) << q      << " * "
        << setw(3) << MAX    << " +"
        << setw(3) << r      << endl;

/* Utilizzo del tipo reale */
   cout << "Scrivere un angolo ,alfa, in radianti\n";
   cin  >> rad;
   seno = sin(rad);
   cout << "Il seno dell' angolo"
        << fixed << setw(9)  << setprecision(3)  << rad
        << " = "
        << fixed << setw(12) << setprecision(11) << seno
        << endl;

/* Utilizzo del tipo carattere */
   cout << "Scrivere un carattere minuscolo\n";
   cin.ignore(4, '\n');
   cin  >> car;
   car = toupper(car);
   cout << "Il carattere maiuscolo corrispondente risulta " << car << endl;
   Attesa();
   return 0;
}
void Attesa(void) {
  cout << "\n\n\tPremere return per terminare";
  cin.ignore(4, '\n');
  cin.get();
}
```

Istruzione di assegnamento



L'istruzione di assegnamento associa un valore ad una variabile.
Sintassi: variabile = espressione**;**
Semantica: Alla variabile viene attribuito il risultato della espressione;
Esempio:

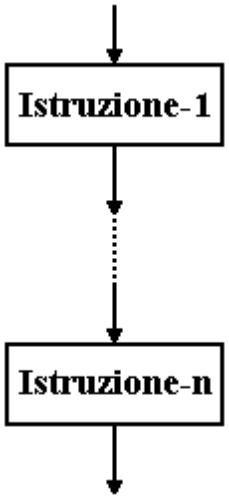
```
int a;
a = 12 * b * 5;
```

Anche un assegnamento ha un valore (il valore assegnato)
Esempio: a = p = 6;
Gli assegnamenti avvengono da destra verso sinistra
Prima esegue l'assegnamento p = 6;
Il risultato di questo assegnamento è 6
Il valore 6 viene assegnato ad a.
Esempio:

```
b = 8;
r = 2;
w = b * (i = b / r);
```

Avremo: i = b / r; i = 4;
w = b * 4; w = 32;

Istruzione composta

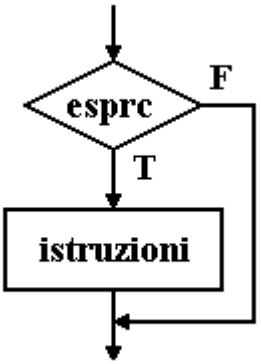


Sintassi:

```
{
    istruzione-1 ;
    . . . . . .
    istruzione-n ;
}
```

Istruzioni condizionali

Istruzione if



Sintassi:

if (esprc **)**
istruzioni-1**;**

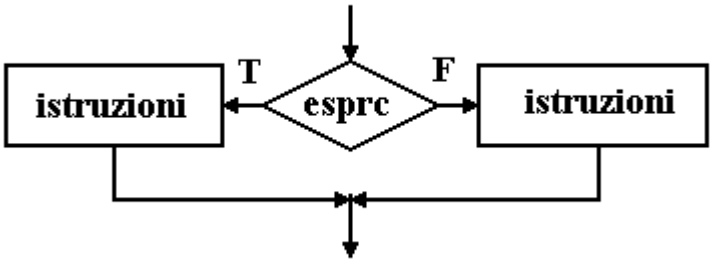
Semantica:

Se la valutazione di esprc restituisce vero allora vengono eseguite le istruzioni-1 altrimenti non vengono eseguite

Esempio:

```
if (i < 100)
  cout << "\nminore di 100";
```

Istruzione if-else



Sintassi:

if (esprc **)**
istruzioni-1**;**
else
istruzioni-2**;**

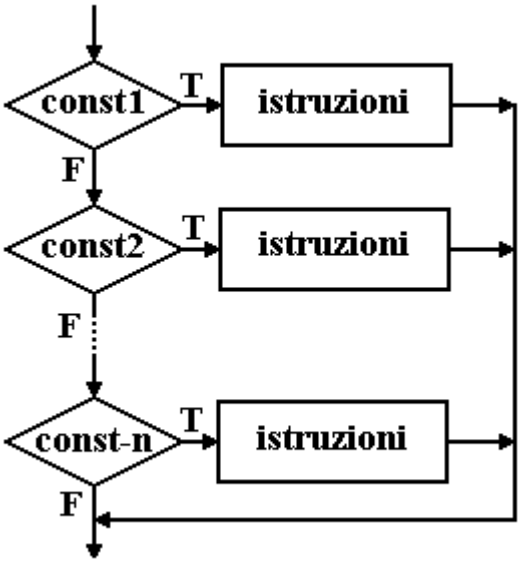
Semantica:

Se la valutazione di esprc restituisce vero allora vengono eseguite le istruzioni-1 altrimenti vengono eseguite le istruzioni-2

Esempio:

```
if (i < 100)
  cout << "\nminore di 100";
else
  cout <<"\nmaggiore o uguale a 100";
```

Istruzione switch-case



Sintassi:

switch (espr**) {**
  case const1**:**
    istruzioni
  case const-n**:**
    istruzioni
  [**default:**
    istruzioni
}

Semantica:

Viene valutata espr, il risultato (intero)viene confrontato con const1, se i due valori sono uguali l'esecuzione inizia dalla istruzione che segue i due punti corrispondenti altrimenti si prosegue confrontando il risultato di espr con il successivo const

Esempio:

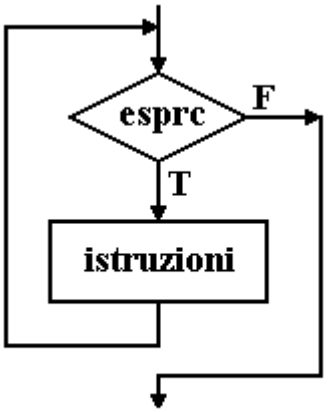
```
switch(x) {
        case '2':
        case '4':
                cout << "\npari";
                break;
        case '1':
        case '3':
                cout << "\ndispari";
                break;
        default:
                cout << "\naltro";
                break;
```

Ritorna

```
/* Switch.cpp
 * Esempio di utilizzo dell'istruzione switch
 */
#pragma hdrstop
#include <condefs.h>
#include <iostream.h>
#include <iomanip.h>
#include <math.h>

//---------------------------------------------------------------------------
#pragma argsused
float ang, fun;
int opr;
void Attesa(void);
int main(int argc, char* argv[]) {
   cout << "Scrivere l'angolo in radianti:  ";
   cin  >> ang;
   cout << "Scrivere la funzione richiesta: \n"
        << "-- 1 -- seno\n"
        << "-- 2 -- coseno\n"
        << "-- 3 -- tangente\n"
        << "-- 4 -- cotangente\n";
   cin  >> opr;

   switch (opr) {
     case 1 : fun = sin(ang);
              cout << "Il seno di "
                   << fixed << setw(8)  << setprecision(3) << ang
                   << " ="
                   << fixed << setw(10) << setprecision(5) << fun
                   << endl;
              break;
     case 2 : fun = cos(ang);
              cout << "Il coseno di "
                   << fixed << setw(8)  << setprecision(3) << ang
                   << " ="
                   << fixed << setw(10) << setprecision(5) << fun
                   << endl;
              break;
     case 3 : fun = cos(ang);
              if (fun == 0.0)
                 cout << "Valore della tangente infinito\n";
              else {
                 fun = sin(ang)/ fun;
                 cout << "La tangente di "
                      << fixed << setw(8)  << setprecision(3) << ang
                      << " ="
                      << fixed << setw(10) << setprecision(5) << fun
                      << endl;
              }
              break;
     case 4 : fun = sin(ang);
              if (fun == 0.0)
                 cout << "Valore della cotangente infinito\n";
              else {
                 fun = cos(ang)/ fun;
                 cout << "La cotangente di "
                      << fixed << setw(8)  << setprecision(3) << ang
                      << " ="
                      << fixed << setw(10) << setprecision(5) << fun
                      << endl;
              }
              break;
     default: cout << "Funzione non prevista\n";
   }
   Attesa();
   return 0;
}
void Attesa(void) {
  cout << "\n\n\tPremere return per terminare";
  cin.ignore(4, '\n');
  cin.get();
}
```

Istruzioni iterative

Istruzione while



Sintassi:

while (esprc**)**

istruzioni**;**

Semantica:

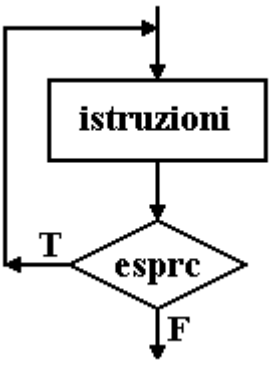
Viene verificata espr se risulta vera viene eseguita l'istruzione .

Il ciclo si ripete fintantoché esprc risulta vera.

Esempio:

```
i = 1;
while (i < 5){
        cout << "\nciclo num " << i;
        i++;
}
```

Istruzione do - while



Sintassi:

do {

istruzioni**;**

} while(esprc**);**

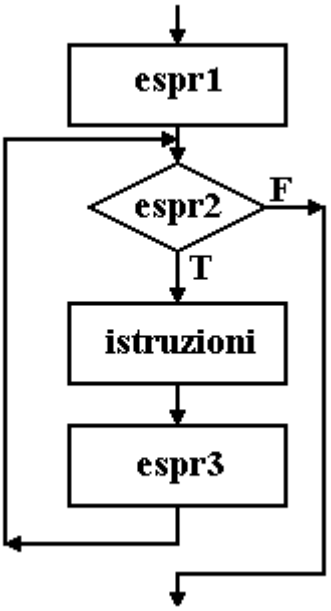
Semantica:

Viene eseguita l'istruzione, successivamente viene controllata esprc se risulta vera il ciclo viene ripetuto.

Esempio:

```
i = 1;
do {
        cout << "\nciclo num "<< i;
        i++;
} while (i < 5);
```

Istruzione for



Sintassi:

for (espr1**;** espr2**;** espr3**)**
        istruzioni**;**

Semantica:

Il ciclo inizia con l'esecuzione di espr1 (viene eseguita solo la prima volta)
Quindi viene valutata espr2; se risulta vera viene eseguita istruzione, al termine viene eseguita espr3 e di nuovo valutata espr2.
Il ciclo si ripete fintanto che espr2 risulta vera.

Esempio:

```
for (i = 1; i < 5; i++)
        cout << "\nciclo num "<< i;
```

Esempio:

```
for ( i = 0, j = strlen(s) - 1; i < j; i++, j--)
        c = s[i], s[i] = s[j], s[j] = c;
```

Inverte una stringa di caratteri.

Esempio:

```
for ( ; ; )
```

Crea un ciclo infinito

Ritorna

```
/* Cicli.cpp
 * Esegue la motiplicazione tra numeri interi anche negativi
 * utilizzando le istruzioni while, do...while, for
 */
#pragma hdrstop
#include <condefs.h>
#include <iostream.h>
#include <iomanip.h>

//---------------------------------------------------------------------------
#pragma argsused
void Attesa(void);
int main(int argc, char* argv[]) {
  int x, y, z, u;

  /* Moltiplicazione tra due numeri naturali */
  cout << "Valore del moltiplicando  = ";
  cin  >> y;
  cout << "Valore del moltiplicatore = ";
  cin  >> x;

  /* Uso dell'istruzione while */
  z = 0;
  if(x<0)
    u = -x;
  else
    u = x;
  while (u > 0) {
     z += y;
     u -= 1;
  }
  if (x < 0) z = -z;
  cout << setw(5) << y << " * "
       << setw(5) << x << " = "
       << setw(5) << z << endl;

  /* Uso dell'istruzione do ... while */
  z = 0;
  if(x<0)
    u = -x;
  else
    u = x;
  if (u != 0)
     do {
        z += y;
        u -= 1;
     } while (u > 0);
  if (x < 0) z = -z;
  cout << setw(5) << y << " * "
       << setw(5) << x << " = "
       << setw(5) << z << endl;

  /* Uso del ciclo for */
  z = 0;
  if(x<0)
    u = -x;
  else
    u = x;
  for(u; u > 0; u--)
      z +=y;
  if (x < 0) z = -z;
  cout << setw(5) << y << " * "
       << setw(5) << x << " = "
       << setw(5) << z << endl;

  Attesa();
  return 0;
}
void Attesa(void) {
  cout << "\n\n\tPremere return per terminare";
  cin.ignore(4, '\n');
  cin.get();
}
```

Istruzione break

L'istruzione **break** provoca l'uscita incondizionata da un ciclo
Esempio:

```
n = 0;
for ( i = 0; i < 10; i++)
{
        cin  >> n;
        if (n == 0)
                break;
        cout << endl << n+n;
}
```

Soluzione alternativa senza l'istruzione break (da preferire)

```
n = 1;
for ( i = 0; i < 10 && n != 0; i++)
{
        cin  >> n;
        if (n != 0)
                cout << endl << n+n;
}
```

Istruzione continue

L'istruzione **continue** forza l'inizio dell'iterazione successiva di un ciclo, provoca l'esecuzione immediata della parte di controllo del ciclo.
Esempio:

```
for (i = 1; i < 10; i++)
{
        if ( i == 7)
                continue;
        cout << endl << i;
}
```

Soluzione alternativa senza l'istruzione continue (da preferire)

```
for (i = 1; i < 10; i++)
{
        if ( i != 7)
                cout << endl << i;
}
```

Istruzione goto

Le istruzioni **goto** e **label** sono consentite dal linguaggio ma sono da evitare.

Vettori

Il C mette a disposizione un costruttore di tipo detto vettore (o array) che consente di raggruppare variabili dello stesso tipo in un unico oggetto.

Sintassi: tipo nome [num-comp];

tipo	è il tipo dei valori contenuti in ciascuna variabile.
nome	consente di accedere ad ognuna delle variabili di cui è costituito.
num-comp	indica il numero di variabili che fanno parte del vettore.
	num-comp deve essere una espressione costante

Esempio:

```
const int a = 4;
int vet[a];
int vet[5];
```

Ogni variabile è accessibile tramite il nome, comune a tutte, specificandone la sua posizione (l'indice) all'interno del vettore.

In C la prima variabile ha indice zero è quindi accessibile come nome[0]; le successive sono accessibili come nome[1], nome[2], ... nome[num-comp - 1].

Un vettore può essere visto come un'area di memoria che contiene consecutivamente tutte le variabili che lo compongono.

Esempio:

```
float temperatura[7];
temperatura[0] = 27.0;
temperatura[6] = 5.4;
```

Un vettore può essere inizializzato nel modo seguente: float vett[] = {17.5, 43.4, 32.9, 56.1 };

La dimensione del vettore è uguale al numero di valori introdotti.

Esempi di definizioni;

```
#define MAX 12
const int n = 12;
float vet1[MAX];
float vet2[n];
float vet3[12];
```

I tre vettori definiti sono simili, l'importante che tra le parentesi quadre vi sia una espressione costante.
Un vettore può avere più dimensioni, cioè la variabile base di un vettore può essere a sua volta un vettore.

Esempio: int matr[5][4];
matr risulta un vettore di cinque vettori di quattro interi.
Il concetto matematico corrispondente è la matrice.

Ritorna

```
/* Vettori.cpp
 * Esempio di inizializzazione di vettori
 */
#pragma hdrstop
#include <condefs.h>
#include <iostream.h>
#include <iomanip.h>
#define nv 10
//---------------------------------------------------------------------------
#pragma argsused
void Attesa(void);

int main(int argc, char* argv[]) {
  int vet[nv];
  int i;

  for(i = 0; i < nv; i++)
     vet[i] = i*i;

  for(i = nv-1; i >= 0; i--)
     cout << setw(5) << vet[i];

  Attesa();
  return 0;
}
void Attesa(void) {
  cout << "\n\n\tPremere return per terminare";
  cin.get();
}
```

Costanti

Sono dei valori espressi direttamente nel codice del programma.

Intere

decimali		Esempio:	23
ottali	iniziano con '0'	Esempio:	027
esadecimali	iniziano con '0x'	Esempio:	0x17
long	terminano con 'l' oppure ‘L’	Esempio:	56732L
unsigned	terminano con 'u' oppure 'U'	Esempio:	45238U

Reali

Sono costituite da:

- parte intera in forma decimale
- punto decimale
- parte frazionaria in formato decimale
- un carattere 'e' oppure 'E'
- un esponente
- Esempio: 34.28E7

float	se terminano con 'f' oppure 'F'	Esempio:	25.63F
long	se terminano con 'l' oppure 'L'	Esempio:	56.71L

Carattere

Sono indicate fra apici
L’insieme di caratteri ammessi è quello definito dal codice della macchina (ASCII, EBCDIC)
Esempio ‘g’

Caratteri speciali:

'\a'	suono
'\b'	backspace
'\f'	salto pagina
'\n'	new line
'\r'	ritorno ad inizio riga
'\t'	tabulazione orizzontale
'\v'	tabulazione verticale
'\?'	punto interrogativo
'\\'	backslash
'\''	apice
'\"'	doppio apice
'\0'	NUL
'\xnn'	nn rappresentano cifre esadecimali
'\ooo'	ooo rappresentano cifre ottali

Stringhe

Una stringa è una sequenza di caratteri terminata con \0
Esempio:

"Come è bello il mondo"

Le stringhe, non hanno un limite di lunghezza.

Non è ammesso mettere un a capo prima di aver chiuso la stringa con le virgolette; per proseguire una stringa sulla linea successiva basta chiudere e riaprire le virgolette.

Esempio

```
char frase [] = "Prima parte"
                "Seconda parte"
                "costituiscono una unica stringa"
```

Nota:

'c' è un carattere
"c" è una stringa.

Direttiva #define

Utilizzando la direttiva per il compilatore #define si possono introdurre delle costanti nel programma, il compilatore le sostituirà al nome al momento della compilazione.

Esempi:

```
#define  NUM  12345L      Definisce una costante di tipo log int
#define  NUM  12345U      Definisce una costante di tipo unsigned
#define  NUM  123.4F      Definisce una costante di tipo float
#define  NUM  123.4       Definisce una costante di tipo long double
#define  NUM  O37         Definisce una costante intera espressa in forma ottale
#define  NUM  Ox1f        Definisce una costante intera espressa in forma esadecimale
#define  NUM  'x'         Definisce una costante di tipo carattere
```

Variabili costanti

Una variabile può essere inizializzata in modo costante, non modificabile.
Le variabili const:

- non possono essere modificate
- occupano spazio solo se referenziate
- sono, per default, locali al file

Esempi:

```
const int MAX = 20 + 1;
const char *msg = "Messaggio promozionale";
const  float eps = 1.0e-5;
        Inizializza in modo costante la variabile, di tipo float,  'esp' al valore 1.0e-5
const  char  msg[] = "Attenzione";
        Inizializza in modo costante la variabile, di tipo vettore, 'msg' al valore "Attenzione"
const int size = 100;
double vet[size];        // Variabili const possono essere usate per dimensionare arre di memoria
size = 150;              // ERRORE: variabili const non possono essere modificate
```

L'utilizzo di const rende i programmi

- più leggibili
- più controllati, aumenta la robustezza del codice
- più facili da controllare.

Enumerazione

E' possibile dichiarare un insieme ordinato di costanti con l'operatore **enum.**

Esempio:

enum boolean { FALSE, TRUE }
Il compilatore associa al nome FALSE il valore 0 e al nome TRUE il valore 1.

Esempio:

enum colore {BIANCO = 3, GRIGIO = 4, NERO }
Il compilatore associa
al nome BIANCO il valore 3
al nome GRIGIO il valore 4
al nome NERO il valore 5.

Per i valori non specificati il compilatore continua la progressione a partire dall'ultimo non specificato.
In C++ non è più necessario precedere con enum la definizione di una variabile di tipo enumerato
Esempio:

```
// Definisce un nuovo tipo enumerato
enum semaforo ( verde, giallo, rosso };          verde vale 0, giallo 1, rosso 2
// Definisce una variabile di tipo semaforo
semaforo luce;                                   semaforo diventa un nuovo tipo
// semaforo può essere usato come un int
if(luce == rosso)
        cout << "Fermati!" << endl;
// Converte un int a semaforo
luce = (semaforo) 2;          É necessario un cast per convertire un int a semaforo
Attenzione il cast precedente non fa verifiche
```

Un tipo enumerato equivale a un nuovo tipo
I tipi enumerati sono sempre meglio dei #define perché il compilatore ne verifica il tipo
I tipi enumerati possono rappresentare solo numeri interi (non double o stringhe)
Per definizione occupano il minimo numero di byte per contenere l'intervallo degli interi ammessi, anche se quasi tutti i compilatori li implementano come int (possono sorgere problemi di compatibilità, si può attivare un flag di compilazione per attivarli)
Si può assegnare un valore a ogni significato
Esempio:

```
enum semaforo { rosso = 10; giallo, verde };
        rosso vale 10, giallo 11, verde 12
enum lsemaforo { lrosso='r', lgiallo= 'g', lverde= 'v'};
        accetta anche un char perché equivale a un int
```

Ritorna

```
/*  Costanti.cpp
 *  Costanti- enumerazione
 */
#pragma hdrstop
#include <condefs.h>
#include <iostream.h>
#include <iomanip.h>

//---------------------------------------------------------------------------
#pragma argsused
//#define N   5
const int N = 5;
enum semaforo { ROSSO, VERDE, GIALLO };

int   x, ris;
int   col;
void Attesa(void);
int main(int argc, char* argv[]) {
/* Utilizzo di una costante */
    cout << "Valore di x = ";
    cin  >> x;
    ris = x * N;
    cout << setw(6) << x   << " *"
         << setw(2) << N   << " ="
         << setw(6) << ris << endl;

/* Utilizzo della enumerazione */
    cout << "Scegli un colore indicando il numero corrispondente:\n"
         << "0 -- rosso\n1 -- verde\n2 -- giallo\n";
    cin  >> col;
    if (col == VERDE)
       cout << "Passa l'incrocio\n";
    else
       cout << "Fermati all'incrocio\n";
    Attesa();
    return 0;
}
void Attesa(void) {
  cout << "\n\n\tPremere return per terminare";
  cin.ignore(4, '\n');
  cin.get();
}
```

Operatori

Nella espressione 3 + a * 5

3 a 5 sono gli operandi

+ * sono gli operatori

L'esecuzione delle operazioni indicate è detta valutazione della espressione.

Gli operatori possono essere:

unari es: -a

binari es: a + b

ternari es: (a > b) ? a-b : b-a;

Operatori aritmetici

+	somma
-	sottrazione
*	prodotto
/	divisione
%	modulo (resto della divisione tra interi)

Operatori di autoincremento e autodecremento

++ incrementa la variabile intera di uno

es: b++; equivale ad b = b + 1;

-- decrementa al variabile intera di uno

es: c--; equivale ad c = c - 1;

Possono essere usati nella forma:

prefissa

es: ++a la variabile a prima viene incrementata e poi usata

postfissa:

es a++ la variabile a prima viene usata e poi incrementata

Regole di precedenza:

Permettono di stabilire quale tra gli operatori non separati da parentesi debba venire eseguito per primo.

a + b * c

Prima viene eseguito il prodotto b * c e il risultato verrà sommato ad a

Regole di associatività:

Indicano se una espressione del tipo a op b op c viene considerata come:

(a op b) op c	associatività da sinistra a destra
a op (b op c)	associatività da destra a sinistra

Nel caso di operatori con la stessa precedenza l'associatività condiziona l'ordine di valutazione.

es: !a++

viene valutato !(a++)

NotaBene:

Gli operandi possono essere valutati in qualsiasi ordine

es: a / b / c;

a / b viene eseguito per primo per l'associatività dell'operatore / ma non è stabilito se a è valutato prima di b oppure di c.

es: b[j] = j++;

non è garantito che l'indirizzo di b[j] venga calcolato prima del valore della espressione j++; compilatori diversi possono generare codici diversi.

Regole di precedenza e associatività degli operatori

() [] -> . ::	da sinistra a destra
! ~ - ++ -- + - * & (tipo) sizeof new delete	da destra a sinistra
.* ->*	da sinistra a destra
* / %	da sinistra a destra
+ -	da sinistra a destra
<< >>	da sinistra a destra
< <= > >=	da sinistra a destra
== !=	da sinistra a destra
&	da sinistra a destra
^	da sinistra a destra
|	da sinistra a destra
&&	da sinistra a destra
||	da sinistra a destra
? :	da destra a sinistra
= += -= *= /= %= &= ^= |= <<= >>=	da destra a sinistra
,	da sinistra a destra

Le parentesi modificano le precedenze

Operatore virgola

Due istruzioni separate da una virgola sono eseguite una dopo l'altra ma sono considerate come un'unica istruzione.

Esempio:

```
for (i=0; i<10; i++, k--) { ... }
```

Il valore di due espressioni separate da una virgola è quello della seconda espressione.

Esempio:

```
n = a, b;
```

Il valore assegnato a d n è quello di b.

Esempio:

```
while(isalpha(getchar())
```

Se isalpha è una macro getchar può essere chiamata due volte

mentre nel caso

```
while (c=getchar(), isalpha(c))
```

Le due istruzioni sono lette da sinistra a destra e while decide in base al valore ritornato da isapha(c)

Operatore ternario

Sintassi:

espr1 **?** espr2 **:** espr3

Semantica:

se la valutazione di espr1 restituisce vero allora è uguale a espr2 , altrimenti è uguale espr3

Esempio:

```
a > b ? a : b
```

Esempio:

```
v = x == y ? a*c+5 : b-d
```

Se x è uguale a y a v viene assegnato il valore a*c+5 altrimenti gli viene assegnato b-d

Ritorna

```
/* Operarit.cpp
 * valore delle assegnazioni
 */
#pragma hdrstop
#include <condefs.h>
#include <iostream.h>

//---------------------------------------------------------------------------
#pragma argsused
void Attesa(char *);
int main(void) {
   short num, i;
   int s[5] = { 10, 20, 30, 40, 50 } ;

   cout << "  Assegnamenti:\n";

   num = 10;
   i = 3;
   cout << "num        vale  " << num    << endl;       //valore iniziale 10
   cout << "num--      vale  " << num--  << endl;                        // 10
   cout << "num        vale  " << num    << endl << endl;  //nuovo valore  9

   cout << "--num      vale  " << --num  << endl;                        //  8
   cout << "num        vale  " << num    << endl << endl;  //nuovo valore  8

   cout << "num *= i   vale  " << (num *= i) << endl;                    // 24
   cout << "num        vale  " << num    << endl << endl;  //nuovo valore 24

   cout << "num++; num vale  " << num++
        << ", num      vale  " << num    << endl;                 // 24, 24
   cout << "num        vale  " << num    << endl;          //nuovo valore 25

   cout << "++num; num vale  " << ++num
        << ", num      vale  " << num    << endl;                 // 26, 25
   cout << "num        vale  " << num    << endl;         //nuovo valore  26
   Attesa(" continuare\n");
   num = 32767;
   cout << "Se num     vale  " << num    << endl;                 //  32767
   cout << "++num      vale  " << ++num  << endl << endl;         // -32768

   i = 2;
   cout << "Se i       vale  " << i      << endl;                      // 2
   cout << "Se s[i]    vale  " << s[i];
   cout << " e s[i+1]  vale  " << s[i+1] << endl;                 // 30, 40
   cout << "s[i++]     vale  " << s[i++] << endl;                     // 30
   cout << "Dopo i     vale  " << i << endl << endl;                  //  3
   i = 2;
   cout << "Se i       vale  " << i      << endl;                     //  2
   cout << "Se s[i]    vale  " << s[i];
   cout << " e s[i+1]  vale  " << s[i+1] << endl;                 // 30, 40
   cout << "s[++i]     vale  " << s[++i] << endl;                     // 40
   cout << "Dopo i     vale  " << i      << endl;                     //  3
   Attesa(" terminare");
   return 0;
}
void Attesa(char * str) {
  cout << "\n\n\tPremere return per" << str;
  cin.get();
}
```

Ritorna

```
/* Virgola.cpp
 * Operatori virgola e ternario
 */
#pragma hdrstop
#include <condefs.h>
#include <iostream.h>
//---------------------------------------------------------------------------
#pragma argsused
void Attesa(char *);
int main(void) {
  int a;
  int b = 6;
  int c;

  // Operatore virgola
  a = b * b , c = a - b;
  cout << "c = " << c << endl;

  cout << "Introdurre un numero intero ";
  cin  >> a;

  // Operatore ternario
  c = a > b ? a : b;
  cout << "c = " << c << endl;

  Attesa("terminare");
  return 0;
}
void Attesa(char * str) {
  cout << "\n\n\tPremere return per " << str;
  cin.ignore(4, '\n');
  cin.get();
}
```

Operatori relazionali

Gli operatori relazionali consentono di effettuare un confronto tra due oggetti dello stesso tipo secondo una relazione di ordine definita.

Se il confronto è vero restituiscono il valore 1 altrimenti 0.

> **maggiore**

es: 5 > 5 risulta 0

>= **maggiore-uguale**

es: 5 >= 5 risulta 1

< **minore**

es: 5 < 5 risulta 0

<= **minore-uguale**

es: 5 <= 5 risulta 1

== **uguale**

es: 5 == 5 risulta 1

!= **non uguale**

es: 5 != 5 risulta 0

Operatori logici

Gli operatori logici effettuano le operazioni booleane di espressioni con valore di verità.

&& **and**

Restituisce 1 solo se entrambe le espressioni sono vere, 0 altrimenti

es: (5 > 3) && (4 == 2) risulta 0

Nota: Se la prima espressione è falsa la seconda espressione non viene valutata in quanto il risultato è comunque falso.

|| **or**

Restituisce 0 solo se entrambe le espressioni sono false, 1 altrimenti

es: (5 > 3) || (4 == 2) risulta 1

Nota: Se la prima espressione è vera la seconda espressione non viene valutata in quanto il risultato è comunque vero.

! **not**

Restituisce 1 se l'espressione è 0 e viceversa.

es: !(5 > 3) risulta 0

Ritorna

```
/* Operel.cpp
 * Esempio con operatori relazionali e logici
 */
#pragma hdrstop
#include <condefs.h>
#include <iostream.h>

//---------------------------------------------------------------------------
#pragma argsused
void Attesa(char * str);
int main(int argc, char* argv[]) {
   int a, b, c;
   char ch1, ch2;

   /* Confronto tra interi */
   cout << "Introdurre il valore di a:  ";
   cin  >> a;
   cout << "Introdurre il valore di b:  ";
   cin  >> b;
   cout << "Introdurre il valore di c:  ";
   cin  >> c;
   cout << "a > b                 ?  " << (a > b)               << endl;
   cout << "c <= b                ?  " << (c <= b)              << endl;
   cout << "(a == b) && (c < a)   ?  " << ((a == b) && (c < a)) << endl;
   cout << "(a != c) || (b > c)   ?  " << ((a != c) || (b > c)) << endl;
   cout << "!(a >= c)             ?  " << (!(a >= c))           << endl;
   Attesa(" continuare\n");

   /* Confronto tra caratteri */
   cout << "Introdurre un carattere dell'alfabeto     ch1:   ";
   cin  >> ch1;
   cout << "Introdurre un'altro carattere alfabetico  ch2:   ";
   cin.ignore(4, '\n');
   cin  >> ch2;
   cout << "ch1 < ch2  (" << ch1 << " < " << ch2 << ")    ?  "
        << (ch1 < ch2) << endl;
   Attesa(" terminare");
   return 0;
}
void Attesa(char * str) {
  cout << "\n\n\tPremere return per" << str;
  cin.ignore(4, '\n');
  cin.get();
}
```

Conversioni di tipo

Quando un operatore ha operandi di tipo diverso vengono convertiti in un tipo comune; trasformando l'operatore più piccolo in quello più grande(rispetto alla memoria occupata).

I char possono essere usati in qualsiasi espressione aritmetica (come int), ma attenzione che quando è convertito in intero può risultare negativo (dipende dalla macchina).

Una variabile double ha almeno dieci cifre significative, mentre un tipo float ne ha sei.
La posizione del punto decimale non ha importanza ai fini della precisione:

i numeri 3.14, 314.0, 0.314, 314000000.0, 0.00000314
hanno tutti tre cifre significative.

Regole di conversione aritmetiche

- Se c'è un operando long double l' altro viene convertito in long double.
- altrimenti se c'è un float l'altro viene convertito in double
- altrimenti se c'è un float l'altro viene convertito in float
- altrimenti se c'è un long l'altro viene convertito in long
- altrimenti i char e gli short vengono convertiti in int.

Negli assegnamenti il valore del lato destro viene trasformato nel tipo del lvalue.
E' possibile forzare una conversione tramite l'operatore cast
Esempio:

```
sqrt((double) n);
```

Converte n in double prima di passarlo alla funzione sqrt
Gli argomenti di una funzione al momento della chiamata sono convertiti nel tipo dichiarato nel prototipo della funzione.

Gerarchia dei tipi predefiniti

long double, double, float, unsigned long, long, insigned int, int
Gli operandi char, shor int con segno vengono sempre convertiti in int prima di effettuare qualsiasi operazione.

Nota: Il risultato di una operazione del tipo 7 / -2 può dare risultati diversi dipendenti dalla macchina può dare

-3 la parte intera del quoziente troncato verso 0 oppure
-4 massimo intero non superiore al quoziente

con la conseguente possibilità di avere

7 % -2 uguale a -1 oppure uguale ad 1

Per garantire la portabilità dei programmi può essere necessario complicare le espressioni:

scrivendo -(7 / 2)
ricorrendo ad espressioni condizionali
utilizzando le funzioni di libreria div ed ldiv in cui vale sempre la relazione

numeratore = quoziente * denominatore + resto

Ritorna

```
/* Convers.cpp
 * Esempi di conversione di tipo
 */

#pragma hdrstop
#include <condefs.h>
#include <iostream.h>

//---------------------------------------------------------------------------
#pragma argsused
void Attesa(char * str);
int main(int argc, char* argv[]) {
  int    a = 556;
  short  b = 25;
  char   c = 'g';
  float  d = 135.659;
  double e = 23.56e34;

  int f;
  float g;
  double h;

  f = c;
  cout << "f = c               f --> " << f << endl;
  c = f + 4;
  cout << "c = f + 4           c --> " << c << endl;
  c = (char) 89;
  cout << "c = (char) 89       c --> " << c << endl;
  g = a / b;
  cout << "g = a / b           g --> " << g << endl;
  g = (float) a / b;
  cout << "g = (float) a / b   g --> " << g << endl;
  g = d / b;
  cout << "g = d / b           g --> " << g << endl;
  h = e * d;
  cout << "h = e * d           h --> " << h << endl;
  Attesa(" terminare");
  return 0;
}
void Attesa(char * str) {
  cout << "\n\n\tPremere return per" << str;
  cin.get();
}
```

Output formattato

Le funzioni printf forniscono le conversioni per l'output formattato.

int **fprintf**(FILE *stream, const char *format, ...)

fprintf converte e scrive l'output su stream, sotto il controllo del parametro format.
Il valore restituito è il numero di caratteri scritti, oppure un valore negativo in caso di errore.

La stringa di formato contiene due tipi di oggetti: caratteri normali, che vengono copiati sullo stream di output, e specifiche di conversione, ognuna delle quali provoca la conversione e la stampa del successivo argomento di fprintf.
Ogni specifica di conversione inizia con il carattere % e termina con il carattere di conversione.
Tra questi due caratteri si possono inserire, nell'ordine:

Un flag (in un ordine qualsiasi), che modificano la specifica di conversione:

- `-` che specifica un incolonnamento destro dell'argomento all'interno del suo campo.
- `+` che specifica che il numero verrà sempre stampato insieme al suo segno.
- spazio se il primo carattere non è un segno, verrà stampato uno spazio.
- `0` per le conversioni numeriche,
  specifica un incolonnamento in cui il numero è eventualmente preceduto da un opportuno numero di zeri.
- `#` che specifica un formato alternativo dell'output.
  - con o — la prima cifra è uno zero.
  - con x o X — un risultato non nullo viene preceduto con il suffisso 0x o 0X.
  - con e, E, f, g, G — l'output ha sempre il punto decimale;
  - con g, G — eventuali zeri non significativi non vengono rimossi.

Un numero che specifica un'ampiezza minima del campo.
L'argomento convertito viene stampato in un campo di ampiezza almeno pari a quella specificata.
Se l'argomento convertito ha meno caratteri dell'ampiezza del campo, esso viene incolonnato a destra (o a sinistra, se è stato richiesto un incolonnamento sinistro) in modo da raggiungere l'ampiezza opportuna.
Il carattere usato per incolonnare è, di norma, lo spazio bianco, ma diventa 0 se è stato fornito il flag opportuno.

Un punto che separa l'ampiezza del campo dalla precisione.

Un numero, per la precisione
che specifica il massimo numero di caratteri stampabili da una stringa,
oppure il numero di cifre da stampare dopo il punto decimale in presenza dei flag e, E, f,
oppure il numero di cifre significative per i flag g, G,
oppure il numero minimo di cifre da stampare per un intero (per raggiungere l'ampiezza desiderata verranno aggiunti degli zeri iniziali).

Un modificatore di lunghezza h oppure l (lettera elle) oppure L;

- "h" indica che l'argomento corrispondente dev'essere stampato come short o come unsigned short;
- "l" indica che esso dev'essere stampato come long o come unsigned long;
- "L" indica che l'argomento è un long double.

L'ampiezza o la precisione, o anche entrambe, possono essere indicate con *, nel qual caso il loro valore viene calcolato convertendo l'argomento (o gli argomenti) successivo, che dev'essere di tipo int.
Se il carattere che segue % non è un carattere di conversione, il comportamento è indefinito.

int **printf**(const char *format, ...)

printf (....) equivale a fprintf(stdout, ...).

TABELLA CONVERSIONI DI PRINTF

CARATTERE	TIPO DELL'ARGOMENTO; CONVERTITO IN
d, i	int; notazione decimale con segno
o	int; notazione ottale priva di segno (senza zero iniziale)
x, X	int; notazione esadecimale priva di segno (senza 0x oX iniziale), usando abcdef per 0x e ABCDEF per 0X
u	int; notazione decimale priva di segno
c	int; carattere singolo, dopo la conversione a unsigned char
s	char *; stampa caratteri dalla stringa fino al raggiungimento di '\0' o della precisione
f	double; [-]m.dddddd, dove il numero delle d è dato dalla precisione (il default è 6).
e, E	double; la notazione decimale prevede il formato [-] m.ddddddde±xx oppure [-]m.dddddd±Exx, dove il numero delle d è dato dalla precisione (default è 6). La pre cisione 0 sopprime il punto decimale.
g, G	double; usa %e oppure %E se l'esponente è minore di -4 o maggiore o uguale alla precisione; altrimenti usa %f. Gli zeri superflui non vengono stampati.
p	void *; puntatore (rappresentazione dipendente dall'implementazione).-
n	int *; il numero dei caratteri stampati fino a questo momento in questa chiamata a printf viene scritto nell'argomento. Nessun argomento viene convertito.
%	non converte alcun argomento; stampa un %.

int **sprintf**(char *s, const char *format, ...)

sprintf è uguale a printf, ma stampa il suo output nella stringa s, terminata con il carattere '\o'. s dev'essere sufficientemente ampia da potere contenere il risultato.
Il valore restituito non include il carattere di terminazione.

Ritorna

```
/* Printf.cpp
 * Formattazione dell' output con: printf
 */
#pragma hdrstop
#include <condefs.h>
#include <stdio.h>

//---------------------------------------------------------------------------
#pragma argsused
void Attesa(char * str);
int main(int argc, char* argv[]) {
    char ch = 'h', *string = "computer";
    int count = 234, hex = 0x10, oct = 010, dec = 10;
    double fp = 251.7366;

    /* Visualizza interi. */
    printf("=%d=\n=%+d=\n=%X=\n=%x=\n=%o=\n",
           count, count, count, count, count );
    printf("=%6d=\n=%-6d=\n=%06d=\n",
           count, count, count);

    /* Conta i caratteri stampati. */
    printf( "            V\n" );
    printf( "1234567890123%n4567890123456789O\n", &count );
    printf( "Numero di caratteri stampati: %d\n", count );
    Attesa(" continuare");
    /* Visualizza caratteri. */
    printf( "=%10c=\n=%5c=\n",
            ch, ch );

    /* Visualizza stringhe. */
    printf( "=%25s=\n=%25.5s=\n",
            string, string );

    /* Visaulizza numeri reali. */
    printf( "=%f=\n=%.2f=\n=%15.5f=\n=%e=\n=%E=\n",
            fp, fp, fp, fp, fp );

    /* Visualizza con differenti radici. */
    printf( "%i    %i    %i\n",
            hex, oct, dec );

    /* Attenzione alle conversioni */
    printf("%d     %d\n", 45678.3 , ch);
  Attesa(" terminare");
  return 0;
}
void Attesa(char * str) {
  printf("\n\n\tPremere return per%s", str);
  fflush(stdin);
  getchar();
}
```

Input formattato

Le funzioni scanf gestiscono la conversione di input formattato.

int **fscanf**(FILE *stream, const char *format, ...)

fscanf legge da stream sotto il controllo di format, ed assegna i valori convertiti ai suoi successivi argomenti, ognuno dei quali dev'essere un puntatore.
Essa termina quando ha esaurito format. fscanf restituisce EOF in caso di raggiungimento della fine del file o di errore prima di qualsiasi conversione; altrimenti, restituisce il numero di elementi di input convertiti ed assegnati.

Normalmente la stringa di formato contiene le specifiche di conversione, che vengono utilizzate per interpretare l'input.

La stringa di formato contiene:

Spazi bianchi	che vengono ignorati.
Caratteri normali	(escluso %), che ci si aspetta concordino con i successivi caratteri non bianchi dello stream di input.
Specifiche	di conversione, composte da %, un carattere * opzionale di soppressione dell'assegnamento, un numero opzionale che specifica un'ampiezza massima del campo, un carattere opzionale h, 1 o L che indica la dimensione del target ed un carattere di conversione.

Una specifica di conversione determina la conversione del successivo elemento in input.
Di norma, il risultato viene memorizzato nella variabile puntata dall'argomento corrispondente. Se è presente il carattere di soppressione dell'assegnamento, come in %*s, l'elemento in input viene scartato.
Un elemento in input è definito come una stringa priva di caratteri di spaziatura (tra i quali è compreso il newline); esso si estende fino al successivo spazio o fino al raggiungimento della dimensione del campo, se questa è specificata.
Questo implica che scanf, per trovare il suo input, possa scandire più linee, poiché i newline sono considerati degli spazi. (I caratteri di spaziatura sono gli spazi, i newline, i return, i tab orizzontali e verticali ed i salti pagina).
Il carattere di conversione indica l'interpretazione del campo in input.
L'argomento corrispondente dev'essere un puntatore.
I caratteri di conversione d, i, n, o, u, x possono essere preceduti da h, se l'argomento è un puntatore ad uno short invece che ad un int, oppure da l (lettera elle) se l'argomento è un puntatore a long.
I caratteri di conversione e, f, g possono essere preceduti da l se l'argomento è un puntatore a double e non a float, e da L se è un puntatore a long double.

int **scanf**(const char *format, ...)

scanf (....) equivale a fscanf(stdin, ...).

int **sscanf**(char *s, const char *format, ...)

sscanf (s, ...) equivale a scanf (. . . .) , ad eccezione del fatto che l'input viene prelevato dalla stringa s.

CONVERSIONI DI SCANF

CARATTERE	DATI IN INPUT; TIPO DELL'ARGOMENTO
d	intero decimale; int *
i	intero; int *. L'intero può essere in ottale (preceduto da uno 0) oppure in esadecimale (preceduto da 0x oppure 0X).
o	intero ottale (preceduto o meno dallo 0); int *
x	intero esadecimale (preceduto o meno da 0x oppure 0X); int
c	caratteri; char *. I prossimi caratteri in input (1 per default) vengono inseriti nella posizione indicata; vengono considerati anche i caratteri di spaziatura; non viene aggiunto il carattere ' \0'; per leggere il prossimo carattere non bianco, usate %1s.
s	stringa di caratteri (non racchiusa fra apici); char *, che punta ad un vettore di caratteri sufficientemente grande da contenere la stringa ed uno '\0' di terminazione, che verrà aggiunto automaticamente.
e, f, g	numero floating-point con segno, punto decimale ed esponente opzionali; float *
p	valore del puntatore, così come viene stampato da printf("%p"); void *
n	scrive nell'argomento il numero di caratteri letti fino a questo momento in questa chiamata; int *. Non viene letto alcun input. Il numero degli elementi letti non viene incrementato.
[...]	trova la più lunga stringa non vuota di caratteri di input corrispondenti all'insieme racchiuso fra le parentesi; char * Aggiunge il carattere '\0'. L'insieme [ ] ...] comprende il carattere].
[^...]	trova la più lunga stringa non vuota di caratteri di input non corrispondenti all'insieme racchiuso fra le parentesi; char *. Aggiunge il carattere '\0'. L'insieme [^] . ..] comprende il carattere].
%	carattere %; non viene effettuato alcun assegnamento.

Ritorna

```
/* Scanf.cpp
 * Illustra l'input formattato:
 *          scanf           fflush
 */
#pragma hdrstop
#include <condefs.h>
#include <stdio.h>

//---------------------------------------------------------------------------
#pragma argsused
void Attesa(char * str);
int main(int argc, char* argv[]) {
    int result, integer;
    float fp;
    char string[81];

    /* Legge dei numeri. */
    printf( "Introdurre un numero intero ed un floating-point: " );
    scanf( "%d%f", &integer, &fp );
    printf( "%d + %f = %f\n\n", integer, fp, integer + fp );

    /* Legge ciascuna parola come una stringa. */
    printf( "Introdurre una frase con quattro parole: " );
    for( integer = 0; integer < 4; integer++ ) {
        scanf( "%s", string );
        printf( "%s\n", string );
    }
    Attesa(" continuare");

    /* Ripolisce il buffer di input e legge con la funzione gets. */
    fflush( stdin );
    printf( "Introdurre la stessa frase (viene letta con gets): " );
    gets( string );
    printf( "%s\n", string );
    fflush(stdin);

    /* Specifica i delimitatori */
    printf( "Introdurre la stessa frase \n"
            "(non verranno considerati i delimitatori tab e new-line: " );
    scanf( "%[^\n\t]s", string );
    printf( "%s\n", string );
    Attesa(" continuare");

    /* Ciclo finche'l'input vale 0. */
    fflush(stdin);
    printf("\n\nIntrodurre un numero in formato decimale (num), hex (0xnum), ");
    printf( "o ottale (0num) .\nIntrodurre 0 per terminare\n\n" );
    do {
       printf( "Introdurre il numero: " );
       result = scanf( "%i", &integer );
       if( result )
          /* Visualizza un intero valido. */
          printf( "Decimale: %i  Ottale: 0%o  Esadecimale: 0x%X\n\n",
                  integer, integer, integer );
       else {
        // Legge un carattere non valido Allora ripulisce il buffer e continua.
          scanf( "%s", string );
          printf( "Numero non valido: %s\n\n", string );
          fflush( stdin );
          integer = 1;
       }
    } while( integer );
  Attesa(" terminare");
  return 0;
}
void Attesa(char * str) {
  printf("\n\n\tPremere return per%s", str);
  fflush(stdin);
  getchar();
}
```

Leggibilità

I seguenti programmi svolgono lo stesso compito ma la leggibilità è evidentemente diversa.

```
/* es1a.c */
#include <iostream.h>
main()
{
        float t, f;
        cin >> t;
        f = t*9/5+32;
        cout << f << endl;
}
```

Rilievi:

Mancanza di commenti

Nome delle variabili non significativi.

Mancanza di indicazione per la richiesta di input.

Output non poco chiaro.

```
/* es1b.c
 * Esegue la conversione tra gradi centigradi e Fahrenheit
 */

#include <iostream.h>
#include <iomanip.h>
main()
{
         float centigradi, fahrenheit;

        cout << "Introdurre i gradi Centigradi :  ";
        cin  >> centigradi;
        fahrenheit = centigradi * 9 / 5 + 32;
        cout << "I gradi Fahrenheit corrispondenti sono -->"
              << fixed << setw(10) << setprecision(5) << fahrenheit << endl;
}
```

I seguenti programmi svolgono lo stesso compito ma la leggibilità è evidentemente diversa.

```
#include <iostream.h>
#include <iomanip.h>
#include <math.h>
double i1,i2,integ;
main() { cout << "Dammi i limiti di integrazione "; cin >> i1 >> i2;
integ= cos(i1)+i1; integ += cos(i2)+i2; integ *= (i2-i1)/2.0;
cout << "L'integrale vale: "<<fixed<<setw(10)<<setprecision(5)<<integ<<endl; }
```

Rilievi:

Mancanza di commenti.

Scrittura delle istruzioni senza indentazione.

Struttura del programma poco chiara.

```
/* Programma di integrazione  es2b.c
 * Esegue l'integrazione della funzione cos(x) + x usando la formula del trapezio
 * Formula risolutiva: Integrale = (f(a)+f(b))*(b-a)/2
 */
#include <iostream.h>
#include <iomanip.h>
#include <math.h>

double lim_inf, lim_sup, integrale;
double valore_inf, valore_sup;

main()
{
        cout << "Scrivere il limite di integrazione inferiore ";
        cin  >> lim_inf;
        cout << "Scrivere il limite di integrazione superiore ";
        cin  >> lim_sup; valore_inf = cos(lim_inf) + lim_inf;
        valore_sup = cos(lim_sup) + lim_sup;
        integrale  = (valore_inf + valore_sup) * (lim_sup - lim_inf) / 2.0;
        cout << "L'integrale vale: "
                << fixed << setw(10) << setprecision(5) << integrale
                << endl; }
```

Alcuni suggerimenti:

- Includere commenti significativi per indicare i punti essenziali degli algoritmi utilizzati.

- Adottare per le variabili dei nomi significativi che richiamino il contenuto e l'utilizzo.

- Dichiarare le variabili in modo distinto, una per riga.

- Scrivere una sola istruzione per riga.

- Indentare opportunamente le istruzioni per evidenziare i blocchi.

- Non usare numeri nei programmi ma definire delle costanti per rendere il programma facilmente modificabile e rendere esplicito il significato.

- Per rendere un programma affidabile e portabile utilizzare i valori nei limiti previsti dai tipi utilizzati.

- Non è buona cosa fare affidamento sulle conversioni automatiche tra i tipi ma è meglio indicare esplicitamente il tipo di dato utilizzato.

- Un programma che fa uso dei caratteri estesi non è portabile.

- Ogni volta che si richiede un input all'utente segnalare con un "prompt" che cosa ci si aspetta indicando i dati necessari.

- Per tutti gli input richiesti controllarne la loro validità.

- Nelle istruzioni condizionali utilizzare comunque le parentesi graffe anche quando si esegue una sola istruzione.

- E` preferibile utilizzare un ciclo for quando le condizioni sono semplici e ben determinate fin dall'inizio.

-Nei cicli evitare per quanto possibile l'utilizzo dell'istruzione "break" ciò può comportare una piccola riduzione di efficienza ma controllando il ciclo in un solo punto è più` facile da capire e da modificare.

- Assolutamente evitare l'istruzione "goto"

- I cicli vanno controllati solo con le istruzioni "while, do .. while, e for ".

Debugger

Il debugger consente di vedere ciò che accade durante l'esecuzione del programma.

I breakpoint

Toggle Breakpoint	F5	Attiva o disattiva un breakpoint nella riga corrente dell'editor di codice

Un breakpoint è un indicatore che chiede al debugger di sospendere l'esecuzione del programma quando viene raggiunta una determinata posizione all'interno del programma.

Un breakpoint può essere inserito solo in una riga che genera del codice.

Quando viene raggiunto un breakpoint, viene richiamato l'ambiente di sviluppo e nel codice sorgente viene evidenziata la riga contenente il breakpoint.

Dopo essersi fermati a un breakpoint è possibile:

- Ispezionare le variabili
- Osservare le chiamate contenute nello stack
- Visualizzare i simboli
- Eseguire passo-passo il codice

I breakpoint possono essere attivati o disattivati a piacere

Un breakpoint semplice provoca la sospensione del programma in esecuzione.

Un breakpoint condizionale provoca la sospensione del programma solo se si verificano determinate condizioni:

- A espressione condizionale: se l'espressione condizionale ha valore true l'esecuzione viene interrotta
- A conteggio di passaggi: l'esecuzione del programma viene sospesa dopo che il breakpoint viene raggiunto per il numero di volte specificato

Con l'opzione View | Breakpoint viene visualizzata la finestra Breakpoint List che contiene per ogni riga:

- Filename: nome del file
- Line: numero di riga
- Condition: condizioni impostate
- Pass: condizioni rguardanti il numero di passaggi impostati

Run to Cursor

Run to Cursor	F4	Avvia il programma e lo esegue fino alla riga in cui è contenuto il cursore.

Il comando Run to Cursor agisce come un breakpoint temporaneo.

Ispezione delle variabili

L'opzione View | Debug Windows | Watches apre la finestra Watch List che consente di ispezionare il valore delle variabili

Nella finestra viene visualizzato il nome della variabile seguito dal suo valore.

La finestra Watch Properties consente di aggiungere o modificare un elemento osservato:

- Expression: specifica il nome della variabile da tenere sotto controllo, accetta anche una espressione matematica
- Repeat count: utile quando si devono ispezionare gli array e si devono tenere sotto controllo solo alcuni elementi dell'array
- Digits: ispezionando un numero in virgola mobile, si specifica il numero di cifre decimali da visualizzare (i numeri vengono arrotondati)
- Enabled: determina l'attivazione dell'elemento
- Opzioni di visualizzazione

E' possibile disattivare gli elementi che non si vogliono ispezionare ma che potranno essere utili in seguito.

Indicazioni abbinate alle variabili:

- Undefined symbol:

  La variabile è fuori del campo di visibilità

- Process not accessible

  Il programma non è in esecuzione

- Disabled

  La variabile è disabilitata

- Variable has been optimized end is not available

  Se si deve ispezionare una variabile soggetta a ottimizzazione la si deve dichiarare "volatile" o disattivare l'opzione Register Variables in Project Opzions → Compiler

Debug Inspector

Inspect	ALT-F5	Apre la finestra Debug Inspect per l'oggetto su cui si trova il cursore

Visualizza il contenuto di oggetti complessi come le classi e i componenti.

E' possibile utilizzarlo solo quando l'esecuzione del programma è sospesa sotto il controllo del debugger

La finestra contiene tre pagine:

- Data: Mostra tutti i dati membro della classe:

  L'elenco dei dati è gerarchico:

  - Gli elementi che appartengono direttamente alla classe
  - I membri della classe da cui dipende questa classe e così via.

  Per ispezionare un dato membro si può fare doppio click sul valore a tale membro e varrà aperta una seconda finestra di debug Inspector

- Methods Visualizza i metodi della classe
  Anche questo elenco è gerarchico:

- Metodi relativi alla classe
- Metodi delle classi da cui questa deriva

- Proprties Mostra le proprietà della classe ispezionata

L'opzione Change consente di modificare il valore di una variabile (da eseguire con estrema attenzione)

L'opzione New Expression consente di immettere una nuovo espressione da ispezionare

L'opzione Show Inherited quando è attivata mostra tutti i dati membro, i metodi e le proprietà della classe ispezionata e di tutte le classi di cui questa classe deriva direttamente o indirettamente.

Finestra Evaluate/Modify

Consente di ispezionare e se necessario modificare il valore di una variabile

Perché la finestra di dialogo possa funzionare è necessario che il programma sia fermo a un breakpoint.

Nota: la finestra non si aggiorna automaticamente quando si esegue un'esecuzione passo-passo nel codice è necessario fare clic sul pulsante Evaluate

View | Call Stack

Visualizza il contenuto dello stack per vedere quali funzioni sono state richiamate dal programma

La finestra Call stack visualizza l'elenco delle funzioni richiamate dal programma e l'ordine in cui sono state richiamate; le funzioni sono elencate in ordine inverso, per prima appare l'ultima funzione eseguita

Finestra CPU View

Consente di visualizzare il programma a livello Assembler

La finestra contiene cinque riquadri:

1. Disassemblaggio
2. Registri
3. Flag
4. Stack
5. Dump

Per poter utilizzare la finestra in modo efficiente è necessario conoscere in modo approfondito il linguaggio Assembler

Ritorna

```
/* Debug.cpp
 * ricerca di un numero in un vettore non ordinato
 */
#pragma hdrstop
#include <condefs.h>
#include <iostream.h>
#define N 10
//---------------------------------------------------------------------------
#pragma argsused
int vet[] = {12, 34, 56, 78, 90, 24, 76, 32, 85, 92};
int x;
int i = 0;
bool tro = false;
void Attesa(char * str);
int main(int argc, char* argv[]) {
  cout << "Numero da cercare ?   ";
  cin  >> x;
 while((i < N) && (!tro)) {
    tro = x == vet[i];
    i++;
 }
 if(tro)
    cout << "Il valore " << x << " e' presente" << endl;
 else
    cout << "Il valore " << x << " non e' presente" << endl;
  Attesa(" terminare");
  return 0;
}
void Attesa(char * str) {
  cout << "\n\n\tPremere return per" << str;
  cin.ignore(4, '\n');
  cin.get();
}
```

2° Modulo

- Funzioni
- Moduli
- Scope
  - Static
- Puntatori
  - E vettori
  - Aritmetica
  - Costanti
  - Dimensione
- Main()
- Cast puntatori
- Strutture
  - Struct
  - Union
  - Typedef
- Memoria
- File
- Preprocessore
  - Inclusione
  - Macro
  - Condizioni
  - Direttive
- Inline
- Operatori sui bit
  - Logici
  - Shift
- Campi bit
- Puntatori a funzione
- Stdarg.h

Funzioni e struttura dei programmi

Un programma è un insieme di definizioni di variabili e di funzioni

Una funzione è un insieme di istruzioni predisposte per svolgere un compito particolare.

Una funzione riceve dei parametri e restituisce un valore.

I parametri vengono sempre passati per valore, ed sono considerati variabili locali alla funzione, sono quindi modificabili all'interno della funzione stessa.

Parametro formale = nome indicato nella definizione della funzione

Parametro attuale = è il valore passato alla funzione quando viene richiamata dal programma.

Gli argomenti passati alla funzione deve essere uguale in numero e dello stesso tipo dei parametri indicati nella definizione.

Se la funzione ritorna un tipo diverso da "int" deve essere dichiarato prima di essere chiamata.

In una funzione può esserci più di una istruzione "return", è però buona norma utilizzarne una sola e posizionarla come ultima istruzione.

Prototipo di una funzione = indica il valore ritornato e quello degli argomenti, il nome dei parametri è facoltativo e può essere diverso da quello usato nella definizione.

Il C non consente di definire una funzione all'interno di un'altra funzione.

Qualsiasi funzione può accedere ad una variabile esterna attraverso il suo nome

E' preferibile evitare conversioni automatiche, nel passaggio dei parametri attuali, usando esplicitamente il tipo previsto.

Se si deve fare una conversione tra dati di tipo diverso è bene farla in modo esplicito utilizzando il cast.

Funzioni ricorsive

Una funzione può richiamare se stessa, in tal caso deve sempre esserci una condizione finale che termina la ricorsione.

Una soluzione ricorsiva è più elegante e compatta ma meno efficiente rispetto ad una soluzione iterativa.

Regole di una buona organizzazione

- Il main dovrebbe solo impostare le variabili globali e richiamare le funzioni principali

- Ogni funzione non dovrebbe superare le trenta istruzioni e dovrebbero essere molto specializzate.

- Scrivere i prototipi delle funzioni all'inizio del programma.

- Il programma va suddiviso tra più funzioni

- Ciascuna funzione deve svolgere una parte definita e indipendente dalla altre.

- La comunicazione tra le funzioni deve essere strettamente controllata, evitare le variabili globali

- Suddividere il programma in macro sottoprogrammi e scriverli in file diversi (raggruppando le funzioni utilizzate per quel dato scopo)

Alcuni esempi a cui prestare attenzione:

```
x = f() + g()
```

La funzione f() potrebbe essere valutata prima di g() o viceversa. Per imporre la sequenza di valutazione memorizzare in variabili ausiliarie i valori intermedi della espressione

```
y = f();
z = g();
x = y + z;
```

Per gli argomenti di una funzione non è specificato l'ordine di valutazione.

```
printf("%d  %d\n", ++n, power(2, n));
```

La funzione power riceve n già incrementato ?

Conviene imporre la sequenza in questo modo:

```
n++;
printf("%d  %d\n", n, power(2, n));
```

Ritorna

```
/* Scontr.cpp
 * Calcola lo scontrino di un acquisto
 */

#pragma hdrstop
#include <condefs.h>
#include <iostream.h>

//---------------------------------------------------------------------------
#pragma argsused

void Attesa(void);
float Tassa(float);

int main(int argc, char* argv[]) {
  float acquisto, amm_tassa, totale;

  // Fase di input
  cout << "\nAmmontare dell'acquisto:  ";
  cin  >> acquisto;

  // Fase di elaborazione
  amm_tassa = Tassa(acquisto);
  totale = acquisto + amm_tassa;

  // Fase di output
  cout << "\nSpesa      =  " << acquisto;
  cout << "\nTassa      =  " << amm_tassa;
  cout << "\nTotale     =  " << totale;
  Attesa();
  return 0;
}
float Tassa(float spesa) {
  float percentuale = 0.065;
  return (spesa * percentuale);
}
void Attesa(void) {
  cout << "\n\n\tPremere return per terminare";
  cin.ignore(4, '\n');
  cin.get();
}
```

Programmazione modulare

Quando le dimensioni di un programma aumentano conviene suddividerlo in più parti (moduli).
Ogni modulo (file sorgente) contiene una sequenza di dichiarazioni di tipi, funzioni, variabili e costanti.
Ciascun modulo deve essere il più possibile indipendente dagli altri moduli e deve svolgere un compito ben definito.
Ogni file sorgente può essere compilato separatamente, i file oggetto sono poi uniti dal linker per produrre un unico file eseguibile.
L'ideale sarebbe che l'unico collegamento tra i vari moduli fosse la chiamata di funzioni.
A volte però ci possono essere dei dati che devono essere accessibili da tutti i moduli.
Per poter usare un nome come riferimento allo stesso oggetto in più file sorgente va definito in un solo file e per tutti gli altri file va dichiarato extern.
Il modo più comune per garantire la coerenza tra i file sorgenti consiste nell'inserire le dichiarazioni in file separati di intestazione (header) e quindi includerli nei file che richiedono tali dichiarazioni.
Poiché un file di intestazione può essere incluso in diversi file sorgenti non deve contenere dichiarazioni che non possono essere duplicate.
Inoltre per evitare più #include dello stesso file si può adottare la seguente tecnica che sfrutta le etichette del preprocessore:
Esempio:

```
// simple.h Prevenire la ridefinizione di oggetti
#ifndef SIMPLE_H
#define SIMPLE_H
#define MAX 45
extern float num;
extern Stampa(char *);
#endif   // SIMPLE_H
```

- se l'etichetta non è definita significa che il file non è stato incluso e in questo caso si deve definire l'etichetta e dichiarare gli oggetti
- se l'etichetta è stata definita significa che gli oggetti sono già stati dichiarati e in questo caso si ignora il codice delle dichiarazioni
- conviene adottare come etichetta il nome del file header scritta in maiuscolo usando il separatore "_"

Vantaggi:

- In caso di modifiche basta ricompilare il file modificato e linkare nuovamente i file oggetto.
- Si possono creare librerie, cioè delle collezioni di funzioni, riutilizzabili per più programmi.
- Raggruppare le dichiarazioni nel file header può garantire contro errori nella dichiarazione delle variabili globali e semplifica le modifiche.

Il linker non riunisce solo il codice oggetto dei vari moduli ma anche quello delle funzioni di libreria usate dal programma.

NotaBene:

L'uso delle variabili globali va limitato al minimo indispensabile.
Raggruppare in un file le funzioni che svolgono all'interno della applicazione una parte comune.
Nel file header conviene mettere anche i prototipi di funzioni e le definizioni di costanti.

Ritorna

```
// Main.cpp
#pragma hdrstop
#include <condefs.h>
#include <iostream.h>
#include "Funz.h"
//---------------------------------------------------------------------------
USEUNIT("Funz.cpp");
//---------------------------------------------------------------------------
#pragma argsused
char *nome = "Semplice ma completo";

void Attesa(char *);
int main(int argc, char* argv[]) {
  funz();
  Attesa("terminare");
  return 0;
}
void Attesa(char * str) {
  cout << "\n\n\tPremere return per " << str;
  cin.get();
}

//---------------------------------------------------------------------------
// Funz.h
//---------------------------------------------------------------------------
#ifndef FunzH
#define FunzH
extern char *nome;
void funz();
//---------------------------------------------------------------------------
#endif

//---------------------------------------------------------------------------
// Funz.cpp
//---------------------------------------------------------------------------
#pragma hdrstop
#include <iostream.h>
#include "Funz.h"
//---------------------------------------------------------------------------
#pragma package(smart_init)
void funz() {
  cout <<  nome <<  '\n';
}
```

Ritorna

```
/* Mcalc.cpp
 * Esecuzione di una espressione in notazione postfissa
 * Esempio:
 * (1 - 2) * (4 + 5)   diventa 1 2 - 4 5 + *
 */
#pragma hdrstop
#include <condefs.h>
#include <iostream.h>
#include <iomanip.h>
#include <math.h>

#include "Calc.h"

//---------------------------------------------------------------------------
USEUNIT("Calc.cpp");

USEUNIT("Getop.cpp");

USEUNIT("Getch.cpp");
//---------------------------------------------------------------------------
#pragma argsused

/* Calcolatrice in notazione Polacca inversa */
int main(int argc, char* argv[]) {
  int type;
  double op2;
  char s[MAXVAL];
  cout << "Utilizzare la notazione postfissa" << endl
       << "<Contrl> Z Per terminare" << endl;
  while ((type = getop(s)) != EOF) {
     switch (type) {
        case NUMBER:
                     push(atof(s));
                     break;
        case '+':
                     push(pop() + pop());
                     break;
        case '*':
                     push(pop() * pop());
                     break;
        case '-':
                     op2 = pop();
                     push(pop() - op2);
                     break;
        case '/':
                     op2 = pop();
                     if(op2 != 0.0)
                        push(pop() / op2);
                     else
                        cout << "Errore: divisione per zero" << endl;
                     break;
        case '\n':
                     cout << "\t"
                          << fixed << setw(8) << setprecision(4) << pop()
                          << endl;
                     break;
        default:
                     cout << "Errore: comando " << s << " sconosciuto" << endl;
                     break;
     }
  }
   return 0;
}
```

Ritorna

```c
/* Calc.h
 * Header della calcolatrice Polacca postfissa
 */
//---------------------------------------------------------------------------
#ifndef CalcH
#define CalcH

#define NUMBER '0'
#define MAXVAL 100

void push(double);
double pop(void);
int getop(char[]);
int getch(void);
void ungetch(int);
//---------------------------------------------------------------------------
#endif
```

Ritorna

```
/* Calc.cpp
 * Gestione della stack per il programma calc.c
 */
//---------------------------------------------------------------------------
#pragma hdrstop
#include <iostream.h>
#include "Calc.h"
//---------------------------------------------------------------------------
#pragma package(smart_init)
int sp = 0;
double val[MAXVAL];

/* push: inserisce f in cima allo stack */
void push(double f) {
  if (sp < MAXVAL)
     val[sp++] = f;
  else
     cout << "Errore: stack pieno; " << f << " non inseribile" << endl;
}

/* pop: preleva e ritorna il valore in cima allo stack */
double pop(void) {
  if (sp > 0)
     return val[--sp];
  else {
     cout << "Errore: lo stack e' vuoto" << endl;
     return 0.0;
  }
}
```

Ritorna

```
/* Getch.cpp
 * legge un carattere per il programma Mcalc.cpp
 */
//---------------------------------------------------------------------------
#pragma hdrstop
#include <iostream.h>
#include "Calc.h"
#define BUFSIZE  100
//---------------------------------------------------------------------------
#pragma package(smart_init)


char buf[BUFSIZE];       /* buffer per ungetch  */
int bufp = 0;            /* la prossima posizione libera in buf[] */

/* getch: preleva un carattere che potrebbe essere stato rifiutato in precedenza */
int getch(void) {
   return (bufp > 0)  ? buf[--bufp]  : cin.get();
}

/* ungetch: rimette un carattere nell'input  */
void ungetch(int c) {
   if (bufp >= BUFSIZE)
       cout << "ungetch: troppi caratteri" << endl;
   else
       buf[bufp++] = c;
}
```

Ritorna

```
/* Getop.cpp
 *  Legge gli operandi nella calcolatrice postfissa
 */
//---------------------------------------------------------------------------
#pragma hdrstop
#include <iostream.h>
#include <ctype.h>
#include "Calc.h"
//---------------------------------------------------------------------------
#pragma package(smart_init)
/* getop: legge il successivo operatore o operando numerico */
int getop(char s[]) {
   int  i, c;

   while((s[0] = c = cin.get()) == ' ' || c == '\t')
       ;
   s[1] = '\0';
   if (!isdigit(c)  &&  c != '.')
       return c;                           /* non e' un numero */
   i = 0;
   if (isdigit(c))                         /* legge la parte intera */
       while (isdigit(s[++i] = c = cin.get()))
           ;
   if (c == '.')                           /* legge la parte frazionaria */
       while (isdigit(s[++i] = c = cin.get()))
           ;
   s[i] = '\0';
   if (c != EOF)
       ungetch(c);
   return NUMBER;

}
```

Le variabili possono essere:

automatiche

Ogni variabile locale ad una funzione viene creata al momento della chiamata della funzione e cessa di esistere quando quest'ultima cessa la sua attività.

static

La variabile locale può conservare il suo valore anche fra una chiamata e l'altra della funzione a cui appartiene.

Una variabile statica definita fuori da tutte le funzioni vale solo per il file sorgente in cui è definita. (Ha un campo di visibilità più limitato di una variabile globale.

Anche le funzioni possono essere definite statiche, risultano utilizzabili solamente nel modulo nel quale sono definite.

Esempio:

```
static int Aggiusta(int rotto);
```

extern

Sono variabili globali e possono essere utilizzate da ogni funzione del programma.

Le variabili esterne sono permanenti cioè mantengono il loro valore anche tra due chiamate di funzione

register

Indica al compilatore di collocare la variabile in un registro (il compilatore può ignorare l'avviso)

L'indirizzo di un registro non può essere conosciuto

Intervallo di visibilità:

Un nome è visibile all'interno dell'intervallo di visibilità e invisibile all'esterno, in C++ questo intervallo è definito staticamente, cioè dipende dal testo del programma e non dalla sua esecuzione.

Lo **scope** di un nome è la posizione di programma all'interno della quale il nome può essere usato.

La visibilità si estende dal punto della dichiarazione fino al termine del blocco in cui si trova la dichiarazione(nome locale).

Se il nome non è dichiarato in un blocco (funzione o classe) la visibilità si estende dal punto della dichiarazione fino al termine del file che contiene la dichiarazione (nome globale).

Esempio:

```
int x = 22;                     // x globale (x1)
void funz(int y)          // Il parametro y è locale a funz
{
        int y;            // Errore y è definita due volte nello stesso spazio di visibilità
        int z = x;                // A z viene assegnato il valore della x globale (x1)
        int x;            // x locale (x2) nasconde x globale
        x = 1;            // 1 è assegnato alla x locale (x2)
        {
                int x;    // x (x3) nasconde il primo x locale
                x = 2;    // 2 è assegnato al secondo x locale (x3)
        }
        x = 3;            // 3 è assegnato al primo x locale (x2)
        ::x = 4;          // Tramite l'operatore scope resolution (::) 4 viene assegnato alla x globale
        (x1)
}
int *p = &x;              // A p viene assegnato l'indirizzo della x globale (x1)
```

Visibilità locale: Una variabile può essere dichiarata dopo qualunque graffa aperta, risulta visibile in quel blocco a partire dal punto della dichiarazione.
Per una variabile automatica dichiarata all'inizio di una funzione lo scope è la funzione stessa.
Il sistema riserva della memoria privata e permanente per una particolare funzione perciò variabili locali aventi lo stesso nome ma dichiarate in funzioni diverse non sono correlate.
Il contenuto di una variabile locale è perso quando la funzione ritorna.
Visibilità di file: un nome dichiarato all'esterno di ogni blocco può essere utilizzato in tutto il file a partire dal punto in cui viene dichiarato.
Una variabile globale può essere mascherata all'interno di una funzione definendo una variabile locale con lo stesso nome.
Le variabili globali sono accessibili a tutte le funzioni, sono da usare con molta cautela all'interno delle funzioni per evitare effetti collaterali difficili da correggere.
Per limitare lo scope di una variabile al file nel quale essa si trova la si definisce "static"
Visibilità globale: un nome dichiarato extern in tutti i file è visibile da tutto il programma
Se la variabile esterna è definita in un file sorgente diverso da quello in cui la si sta usando bisogna dichiararla come "extern" in quest'ultimo file.
Fra tutti i file che costituiscono il programma sorgente uno solo deve contenere la definizione di una variabile esterna, gli altri file possono contenere soltanto dichiarazioni di "extern" che consento di utilizzare le variabili.
E' buona norma definire le variabili in un header file.

Inizializzazione

Nel momento della definizione di una variabile è possibile inizializzarla
Esempio:

```
int  p  =  1;
int  s  = 10 + 8;
int  g  = s * p;
```

Il C++ garantisce che le variabili "extern" ed "static" vengano inizializzate a 0
Le variabili automatiche e register non vengono inizializzate, ed hanno perciò valori indefiniti.
Le variabili locali possono essere inizializzate, ciò verrà fatto ogni volta che la funzione viene chiamata.
Una variabile "static" interna ad una funzione mantiene il suo valore anche fra due chiamate successive della funzione a cui appartiene, perciò il valore sarà nuovamente accessibile quando si rientrerà nella stessa funzione.
Una variabile "static" può essere inizializzata, questa inizializzazione avviene una sola volta all'avviamento del programma, questo consente ad una funzione di riconoscere la prima volta che viene chiamata.

Oggetti

Un oggetto è una zona di memoria
Esempio:

```
int x;                  // x è un oggetto
```

Un lvalue è una espressione che fa riferimento a un oggetto; lvalue originariamente indicava un oggetto che può trovarsi alla sinistra di un assegnamento.
Esempio:

```
x = y + 4;              // x è un lvalue che fa riferimento all'oggetto x
```

Nota: non tutti gli lvalue possono essere usati a sinistra di un assegnamento
Esempio:

```
const int z = 90;
z = 100;                    // Errore z è una costante
```

Definizioni di variabili sparse

In C++ non è più necessario definire tutte le variabili all'inizio di un blocco:
Esempio:

```
void funz(int a)
{
        char s[12];
        strcpy(s, "prova";
        if(a < 5) return;
        int x = 56;             // Le variabili sono visibili dal punto in cui vengono definite
        double d;
        d = x * x;
}
```

Un criterio consigliato è definire una variabile più prossima alla sua inizializzazione.
Il costruttore è richiamato nel momento in cui è definita e quindi viene allocata solo se la si utilizza (esempio dopo il return condizionato).
L'istruzione for supporta una forma speciale per la definizione di variabili.
Secondo lo standard ANSI C++ lo scope delle variabili definite nel for-init statement si estende solo al blocco for
Esempio:

```
for(int i = 0; i < 50; i++)             i è visibile solo nel blocco for
        vett_a[i] += i;
for(i = 0; i < 20; i++)                 ERRORE: i non è più visibile
        vet_b[i] = i;
```

Nei compilatori non aggiornati allo standard ANSI C++ invece la visibilità delle variabili si estende oltre
Esempio:

```
for(int i = 0; i < 10; i++)             i è visibile da qui in poi
        vet[i] = 0;
for(int k = i; k < 20; k++)             k è visibile da qui in poi
        vet[k] = 1;
for(int i = k; i < 30; i++)             ERRORE: ridefinizione di i
        vet[i] = 2;
for(i = k; i < 30; i++)         Funziona i e k sono visibili
        vet[i] = 2;
```

Secondo lo standard ANSI C++ le variabili automatiche costruite all'interno di una condizione hanno scope solo all'interno della condizione e non sono accessibili al di fuori della stessa.
Esempio:

```
if((double d = sqrt(x)) > 10)   d è visibile solo nel blocco if / else
        d += 21;
else
        d *= 3;
d += 45;                                ERRORE: d non è più visibile
```

Ritorna

```
/* Scope.cpp
 * Scope delle variabili
 */
#pragma hdrstop
#include <condefs.h>
#include <iostream.h>
#include <iomanip.h>
//---------------------------------------------------------------------------
#pragma argsused
int somma (int, int);
int diff (int, int);
void Funz1 (int, int);
void Funz2 (int, int);
void Attesa(char *);
int x, y;

int main(int argc, char* argv[]) {
  x = 24;
  y = 9;
  cout << "Inizio     x =" << setw(4) << x << " , y =" << setw(4) << y << endl;
  Funz1(x, y);
  cout << "Dopo Funz1 x =" << setw(4) << x << " , y =" << setw(4) << y << endl;
  Funz2(x, y);
  cout << "Dopo Funz2 x =" << setw(4) << x << " , y =" << setw(4) << y << endl;
  Attesa("terminare");
  return 0;
}
int somma (int a, int b) {
  return(a + b);
}
int diff (int a, int b) {
  return( a - b);
}
void Funz1 (int w, int z) {
  x = somma(w, z);
  y = w - z;
  cout << "In Funz1   x =" << setw(4) << x << " , y =" << setw(4) << y << endl;
}
void Funz2 (int w , int z) {
  int x, y;

  x = w + z;
  y = diff(w, z);
  cout << "In Funz2   x =" << setw(4) << x << " , y =" << setw(4) << y << endl;
}
void Attesa(char * str) {
  cout << "\n\n\tPremere return per " << str;
  cin.get();
}
```

Ritorna

```
/* Static.cpp
 * Esempio di inizializzazione
 */
#pragma hdrstop
#include <condefs.h>
#include <iostream.h>
#include <iomanip.h>
//---------------------------------------------------------------------------
#pragma argsused
int a = 1;
int b = 1;
void f();
void Attesa(char *);
int main(int argc, char* argv[]) {
  while (a < 5 )
    f();
  Attesa("terminare");
  return 0;
}
void f() {
   int c = 1;           // Inizializzata ad ogni chiamata di f()
   static int d = b;
   // Inizializzata solo la prima volta che f() viene chiamata
   b += 2;
   cout << " a =" << setw(3) << a++
        << " b =" << setw(3) << b++
        << " c =" << setw(3) << c++
        << " d =" << setw(3) << d++ << endl;
}
void Attesa(char * str) {
  cout << "\n\n\tPremere return per " << str;
  cin.get();
}
```

Ritorna

```
/* Statcstr.cpp
 * Le stringhe sono in realta' array static
 */
#pragma hdrstop
#include <condefs.h>
#include <string.h>
#include <iostream.h>

//---------------------------------------------------------------------------
#pragma argsused
int Modifica();
void Attesa(char *);
int main(int argc, char* argv[]) {
  while( Modifica() )
     ;
  Attesa("terminare");
  return 0;
}
int Modifica() {
  static unsigned int i = 0;    // Per contare all'interno della stringa
  char *st = "Stringa di tipo static\n";
  if(i < strlen(st)) {          // Conta i caratteri nella stringa
     cout << st;                // Stampa la stringa
     st[i] = 'X';
     i++;                       // Punta al prossimo carattere
     return 1;
  } else
     return 0;                  // Indica che la stringa e' finita
}
void Attesa(char * str) {
  cout << "\n\n\tPremere return per " << str;
  cin.get();
}
/*
 * Le stringhe rappresentano una forma di inizializzazione aggregata, una
 * stringa viene trattata come un array di caratteri.
 * La differenza consiste nel fatto che la stringa e' di tipo static, e quindi
 * se la si modifica i cambiamenti vengono mantenuti tra una chiamata e l'altra
 * delle funzioni
 */
```

Ritorna

```
/* Vardef.cpp
 * Definizioni sparse
 */
#pragma hdrstop
#include <condefs.h>
#include <iostream.h>

//---------------------------------------------------------------------------
#pragma argsused
int index;
void Attesa(char *);
int main(int argc, char* argv[]) {
  int var;
  var = index + 14;      //index è stato inizializzato a zero
  cout << "var      ha il valore " << var << endl;

  int altravar = 13;        //non automaticamente inizializzato

  cout << "altravar ha il valore " << altravar << endl;

  for (int count = 3; count < 6; count++) {
     cout << "count    ha il valore " << count << endl;
     char count2 = count + 65;
     cout << "count2   ha il valore " << count2 << endl;
  }

  static unsigned vars;      //automaticamente inizializzato a zero

  cout << "vars     ha il valore " << vars << endl;
  Attesa("terminare");
  return 0;
}
void Attesa(char * str) {
  cout << "\n\n\tPremere return per " << str;
  cin.get();
}
```

Puntatori

Un puntatore è una variabile che contiene l'indirizzo di un'altra variabile.

Poiché le variabili e il codice si trovano in memoria, ci si può riferire a essi tramite il loro indirizzo iniziale cioè l'indirizzo del primo byte della variabile e del blocco di codice.

Durante la scrittura del codice ci si riferisce agli indirizzi iniziali con nomi simbolici, il compilatore usa nomi simbolici per le funzioni, le variabili globali e le variabili statiche, durante il link i nomi simbolici vengono risolti in indirizzi.

La variabile puntatore denotata con l'operatore * serve per manipolare gli indirizzi.

Esempio: int *pt;

I puntatori permettono si selezionare e manipolare a tempo di esecuzione indirizzi di variabili e di funzioni.

I puntatori hanno quattro impieghi principali

negli array

come argomenti di funzioni

per l'accesso diretto alla memoria

per l'allocazione dinamica della memoria

Ogni puntatore si riferisce ad uno specifico tipo di dato ad eccezione del puntatore a ‘void’.

Il tipo del puntatore dipende dalla variabile a cui punta.

Un puntatore consente di accedere alla variabile a cui punta in quel dato momento, a seconda dell’indirizzo contenuto nel puntatore si può agire su diverse variabili.

Un puntatore deve essere inizializzato prima di essere impiegato, il compilatore non può determinare se un puntatore impiegato in una istruzione è già stato inizializzato.

L'operatore unario ‘&’ fornisce l'indirizzo di un oggetto (non può essere applicato alle espressioni, alle costanti e alle variabili register).

L'operatore ‘*’ di indirezione applicato ad un puntatore accede all'oggetto puntato.

Esempio: double dd;

double *pd = ⅆ

Un puntatore è analogo a qualunque altra variabile: contiene un valore che può essere manipolato dagli operatori, passato come argomento a una funzione, usato come indice in un ciclo ; l'unica differenza è che il valore è trattato come un indirizzo, il suo contenuto è sempre trattato come un valore positivo intero.

Non vi è limite al livello di indirezione

Esempio:

int t;

int *pt = &t;

int **ppt = &pt; avremo che *pt ed **ppt puntano alla stessa variabile.

Puntatori e vettori

Un vettore è una sequenza di variabili dello stesso tipo.

Si possono definire vettori con qualsiasi numero di dimensione (matrici)

Gli indici devono iniziare dalla posizione 0

Un vettore dichiarato static conserva il valore dei suoi elementi tra le chiamate di funzione.

Esempi di inizializzazione:

```
int potenze[4] = {2, 4, 8, 16 };
```

oppure

```
int potenze[] = {2, 4, 8, 16} è implicitamente dichiarato di 4 elementi.
int matr[3][] = {{5, 7}, {8, 10}, {2, 9}};
```

I nomi degli array e i puntatori non sono identici:

- un puntatore è una variabile che contiene un valore
- un array è un identificatore che è usato
  - con le parentesi quadre per manipolare un elemento dell'array
  - da solo per indicare l'indirizzo iniziale dell'array
- array[0] è equivalente a *(array+0)

Qualsiasi operazione effettuabile indicizzando un vettore può essere eseguita tramite puntatori.

```
int  a[10];
int  *p;
p  =  a;
*(p+1) corrisponde ad a[1]
a[1] può essere scritto *(a+1)
```

NotaBene:

Un puntatore è una variabile quindi `p = a` ed `p++` sono istruzioni lecite

mentre il nome di un vettore non è una variabile quindi le istruzioni

`a = p` ed `a++` non sono lecite.

Per definizione aggiungendo 1 ad un puntatore lo si fa puntare all'elemento successivo del vettore, indipendentemente dalla dimensione di ciascun elemento

```
pb = &vet[3];
*(pb+1)                corrisponde a vet[3+1];
*pb+1          corrisponde a vet[3]+1
```

Attenzione a non puntare oltre i limiti, si andrebbe ad alterare la memoria in modo non prevedibile, il C non esegue nessun controllo.

Per la precedenza degli operatori nell'esempio:

g = *ps++;

prima *ps viene assegnato ad g e poi il puntatore viene incrementato all'elemeto successivo.

NotaBene:

float a[5][4];

float *p , *p1;

L'operazione:

p = a[4][0]; non è lecita perché a[4][0] non è un indirizzo

l'operazione corretta è p = &a[4][0]; oppure p = a[4];

Inoltre con p = a[4]; ed p1 = a[3]; avremo che p - p1 vale 4;

Quando il nome di un vettore viene passato ad una funzione viene passata la posizione dell'elemento iniziale cioè un puntatore.

Una funzione non conosce la dimensione di un vettore passato come argomento

Ogni modifica fatta sul vettore nella funzione altera il vettore originale.

Ad una funzione si può passare una parte di un vettore

f(&a[2])

Aritmetica dei puntatori

L'aritmetica dei puntatori dipende dal tipo di puntatori ma tutto viene gestito automaticamente dal compilatore.

L'aritmetica dei puntatori prende sempre in considerazione la dimensione del puntatore; se si aggiunge 1 a un puntatore double esso è incrementato di 8 byte, se invece si sottraggono due puntatori che puntano a due locazioni di memoria adiacenti si ottiene 1 cioè il numero di variabili double tra due puntatori double adiacenti.

La costante 0 può essere assegnata ad un puntatore

p = NULL	(costante definita in stdio.h)

Un puntatore ed un intero possono essere sommati e sottratti

p + n	corrisponde all'indirizzo dell'n-esimo oggetto che segue quello puntato da p
*p++	incrementa di uno il puntatore
(*p)++	incrementa di uno l'oggetto puntato da p

La sottrazione tra puntatori è legale

Se p, q puntano ad elementi dello stesso vettore e p < q allora

q - p +1	è il numero di elementi tra p ed q.
*p++ = val;	val viene assegnato a *p e poi p viene incrementato
val = *--p;	decrementa p e poi assegna a val il valore al quale punta adesso p.

Costanti

La parola chiave const può essere applicata a un puntatore oppure all'area puntata dal puntatore.

Esempio:

```
int a, b;
int *const pa = &a;          // Puntatore const a area non const
pa = &b;                     // ERRORE
*pa = 4;                     // OK

const int *pb = &a;          // Puntatore non const a area const
pb = &b;                     // OK
*pb = 5;                     // ERRORE

const int *const pc = &a;    // Puntatore const a area const
pc = &b;                     // ERRORE
*pc = 6;                     // ERRORE
```

Per passare ad una funzione un argomento che si vuole modificare bisogna passare il puntatore.

Esempio:

```
funz(int *ps)  {
        ps è locale alla funzione
        *ps non è locale alla funzione
    }
```

La parola chiave const può (e deve) essere usata per indicare se parametri e funzioni possono essere modificati.

Esempio:

```
size_t strlen(const char *s);
char *strcpy(char *t, const char *s);
const char *s = "Prova";
strcpy(s, "x");
        // ERRORE: s è un const char* che non può essere il primo parametro di strcpy()

int n;
const int *CRead();
CUpdate(int *cli);
CUpdate(CRead());
        // ERRORE: CRead() restituisce un const int* che non può essere passato a CUpdate()
```

Stringhe

Una stringa viene trattata come un array di caratteri, una stringa è di tipo static, se la si modifica i cambiamenti vengono mantenuti tra una chiamata e l'altra delle funzioni

char cha[] = "stringa";

char *chp = "stringa";

hanno lo stesso effetto viene creata una stringa il cui indirizzo iniziale è usato sia per cha che per chp

che è un puntatore a una variabile, puo' essere modificato chp = cha; e chp++; sono legali

cha[] è un array, cha rappresenta l'indirizzo iniziale dell'array, non è una variabile cha = chp: cha++ non sono lecite.

La dimensione di un puntatore

La dimensione di un puntatore dipende dalla dimensione dell'indirizzo.

Su molti calcolatori la dimensione del puntatore dipende dal tipo di dati, si parla di calcolatori a indirizzo lineare.

I microprocessori Intel 80x86 utilizzano uno schema di memoria a segmenti, l'indirizzo è diviso in due parti: segmento e offset

La dimensione di un puntatore varia secondo il modello di memoria utilizzato dal compilatore, il modello di memoria indica al compilatore la dimensione del codice e dei dati.

Alcuni programmi di piccola dimensione che non necessitano di molto spazio per i dati i segmenti per i puntatori ai dati non cambiano mai, quindi i puntatori possono avere la dimensione dell'offset.

Alcuni programmi invece contengono una grande quantita' di dati e i puntatori ai dati devono spaziare in tutta la memoria, questi programmi devono calcolare il segmento e l'offset quindi la dimensione dei puntatori è maggiore.

Uno dei problemi nell'uso dei diversi modelli di memoria consiste nel fatto che quando si scrive il codice non si conosce il modello di memoria che verra' usato dal compilatore.

Molte implementazioni del C e del C++ permettono di modificare la dimensione di default di un puntatore per una determinata variabile usando le parole chiave near e far (che sono pero' non portabili)

near forza un puntatore che normalmente sarebbe costituito dal segmento e dell'offset ad avere solo l'offset

far forza un puntatore ad essere costituito dal segmento e dall'offset

Per informazioni specifiche rifarsi al manuale del compilatore usato.

Ritorna

```
/* Punt.cpp
 * Utilizzo dei puntatori
 */
#pragma hdrstop
#include <condefs.h>
#include <iostream.h>

//---------------------------------------------------------------------------
#pragma argsused
void Attesa(char *);
int main(int argc, char* argv[]) {
   int var = 9999;
   int *pvar = &var;

   cout << "Puntatori ad una variabile" << endl;
   cout << "var    =  " << var    << endl;     // 9999
   cout << "&var   =  " << &var   << endl;
   cout << "pvar   =  " << pvar   << endl;
   cout << "&pvar  =  " << &pvar  << endl;
   cout << "*pvar  =  " << *pvar  << endl;     // 9999

   Attesa("terminare");
   return 0;
}
void Attesa(char * str) {
  cout << "\n\n\tPremere return per " << str;
  cin.get();
}
```

Ritorna

```
/* Varptrs.cpp
 * Differenti tipi di puntatori a variabili
 */
#pragma hdrstop
#include <condefs.h>
#include <iostream.h>

//---------------------------------------------------------------------------
#pragma argsused
void Attesa(char *);
int main(int argc, char* argv[]) {
  int A;
  int *ap = &A;
  cout << "&A = " << ap << endl;
  unsigned int B;
  unsigned int *bp = &B;   // Il puntatore deve essere dello stesso tipo
  cout << "&B = " << bp << endl;
  long double C;
  long double *cp = &C;
  cout << "&C = " << cp << endl;
  // Si puo' anche ottenere l'indirizzo di una costante
  const int X = 100;
  const int *XP = &X; //XP puo` essere modificato mentre *XP no
  // cout << "&X = " << XP << endl;
  // Non e' possibile assegnare a un puntatore non costante
  // l'indirizzo di una costante
  int * const XPC = &A;    // *XPC puo' essere modificato mentre XPC no
  Attesa("terminare");
  return 0;
}
void Attesa(char * str) {
  cout << "\n\n\tPremere return per " << str;
  cin.get();
}
/*
 * Qualunque tipo di dato puo' avere un puntatore che viene definito sempre
 * nello stesso modo, il nome del tipo di dato, un asterisco e il nome della
 * variabile.
 */
```

Ritorna

```
/* Modify.cpp
 * Una funzione che modifica un parametro attuale
 */
#pragma hdrstop
#include <condefs.h>
#include <iostream.h>

//---------------------------------------------------------------------------
#pragma argsused
void addten(int *);
void Attesa(char *);
int main(int argc, char* argv[]) {
  int x = 37;
  addten(&x);  // Si deve esplicitamente prendere l'indirizzo
  cout << "x = " << x << endl;
  Attesa("terminare");
  return 0;
}
void addten(int *val) {
  *val += 10;
}
void Attesa(char * str) {
  cout << "\n\n\tPremere return per " << str;
  cin.get();
}
/*
 * L'unico modo per modificare un parametro attuale consiste nel passare il
 * suo indirizzo alla funzione.
 * Quando si crea una funzione che modifica il parametro formale e la si
 * inserisce in una libreria, ci si deve assicurare che l'utente sappia che
 * essa richiede un puntatore come argomento
 */
```

Ritorna

```
/* Constarg.cpp
 * Passaggio di un puntatore a un argomento costante
 */
#pragma hdrstop
#include <condefs.h>
#include <iostream.h>

//---------------------------------------------------------------------------
#pragma argsused
void func(const double *);
void Attesa(char *);
int main(void) {
  double A;

  A = 1.2;
  func(&A);
  Attesa("terminare");
  return 0;
}
void func(const double *arg) {
   // Lettura dei valori permessa
   cout << "*arg = " << *arg << endl;
   // *arg = 1.23;  // Scrittura dei valori non permessa
}
void Attesa(char * str) {
  cout << "\n\n\tPremere return per " << str;
  cin.get();
}
/*
 * Nella funzione main() la variabile A modificabile,
 * nella funzione func() invece l'argomento e' di tipo const e
 * quindi non modificabile.
 */
```

Ritorna

```
/* Arrayptr.cpp
 * Un puntatore all'indirizzo iniziale di un array puo' essere
 * trattato come l'array stesso.
  */
#pragma hdrstop
#include <condefs.h>
#include <iostream.h>

//---------------------------------------------------------------------------
#pragma argsused
int c[] = { 1, 2, 3, 4, 5, 6 };
int const sz = 6;
void Attesa(char *);
int main(int argc, char* argv[]) {
  // int *c e' lo stesso di int c[], se c non e' esterna
  int i;
  for(i = 0; i < sz; i++)
     cout << "  c[" << i << "] = " << c[i] << endl;
  Attesa("continuare");
  int *cp = c;
  for(i = 0; i < sz; i++, cp++)
     cout << "  i = " << i << ", *cp = " << *cp << endl;
  Attesa("continuare");
  int a[sz];                           // Alloca lo spazio per 10 interi
  int *b = a;                          // b ed a sono ora equivalenti
  for(i = 0; i < sz; i++)
     b[i] = i * 10;     // Inserisce alcuni valori
  for(i = 0; i < sz; i++)
     cout << "  a[" << i << "] = " << a[i] << endl;
  // a[i] contiene gli stessi valori che sono stati inseriti in b[i]
  Attesa("terminare");
  return 0;
}
void Attesa(char * str) {
  cout << "\n\n\tPremere return per " << str;
  cin.get();
}
/*
 * In C++ ogni volta che si calcola, a tempo di esecuzione, l'indirizzo
 * di una variabile, si usa un puntatore;
 * un array e dunque un particolare tipo di puntatore.
 * I nomi degli array e puntatori non sono identici:
 * un puntatore é' una variabile che contiene un valore,
 * mentre un nome di array e' un identificatore
 * che e' usato con le parentesi quadre per manipolare un elemento di un array
 * e da solo per indicare l'indirizzo iniziale di aun array.
 * Non e' possibile modificare il nome di un array
 */
```

Ritorna

```
/* Index.cpp
 * Dimostrazione dell'indicizzazione degli array
 */
#pragma hdrstop
#include <condefs.h>
#include <iostream.h>

//---------------------------------------------------------------------------
#pragma argsused
void Attesa(char *);
int main(int argc, char* argv[]) {
  int x[10];
  int i;
  for(i = 0; i < 10; i++)
     x[i] = 100 - i * i;
  // Tre differenti modi di selezionare l'elemento zero
  cout << "x[0]     = " << x[0]     << endl;
  cout << "*(x + 0) = " << *(x + 0) << endl;
  cout << "*x       = " << *x       << endl << endl;
  // Stampa l'array usando l'addizione dei puntatori
  for( i = 0; i < 10; i++)
     cout << "*(x + " << i << ") = " << *(x + i) << endl;
  Attesa("terminare");
  return 0;
}
void Attesa(char * str) {
  cout << "\n\n\tPremere return per " << str;
  cin.get();
}
/*
 * Gli indici devono iniziare alla posizione 0
 * Gli array sono indicizzati, a tempo di esecuzione, prendendo l'indirizzo
 * iniziale e aggiungendo il numero di byte necessario per ottenere l'indirizzo
 * dell'elemento desidarato
 */
```

Ritorna

```
/* Array_p.cpp
 * Esempio in cui un array viene passato come parametro ad una funzione
 */
#pragma hdrstop
#include <condefs.h>
#include <iostream.h>

//---------------------------------------------------------------------------
#pragma argsused
const sz = 8;
// associa il nome numList al tipo di dato: un array di otto interi
typedef int numList[sz];

void add (numList, int);
void Attesa(char *);
int main(int argc, char* argv[]) {
  // definisce un array (si noti la forma d'inizializzazione)
  numList  numeri = {12, 3, 4, 5, 6, 7, 8, 22};

  cout << "Valore degli elementi dell'array: " << endl;
  int i;
  for (i = 0; i < sz; ++i)
     cout << i << ". " << numeri [i] << endl;

  // somma 5 ad ogni elemento dell'array
  add (numeri, 5);

  cout << endl << "Valore finale degli elementi dell'array: " << endl;
  for (i = 0; i < sz; ++i)
     cout << i << ". " << numeri [i] << endl;
  Attesa("terminare");
  return 0;
}
void add (numList nl, int n) {
  for (int i = 0; i < sz; ++i)
     nl [i] += n;
}
void Attesa(char * str) {
  cout << "\n\n\tPremere return per " << str;
  cin.get();
}
```

Ritorna

```
/* Ptrarit.cpp
 * Dimostrazione della aritmetica dei puntatori
 */
#pragma hdrstop
#include <condefs.h>
#include <iostream.h>

//---------------------------------------------------------------------------
#pragma argsused
void Attesa(char *);
int main(int argc, char* argv[]) {
  char A   = 'a';
  char *cp = &A;       // Crea e inizializza un puntatore a char
  double d   = 1.119;
  double *dp = &d;     // Crea e inizializza un puntatore double

  cout << "Addizione & Sottrazione: " << endl << endl;
  char   *cp2 = cp + 1;
  double *dp2 = dp + 1;
  cout << "cp2 = cp + 1; dp2 = dp + 1; "<< endl;
  cout << "cp2 - cp = " << (cp2 - cp) << endl;
  cout << "dp2 - dp = " << (dp2 - dp) << endl;
  cout << "(int)cp2 - (int)cp = " << ((int)cp2 - (int)cp) << endl;
  cout << "(int)dp2 - (int)dp = " << ((int)dp2 - (int)dp) << endl;
  cout << endl << "Incremento & Decremento:" << endl << endl;
  cout << "cp2 = cp; dp2 = dp; cp2--; cp2--; dp2++; dp2++; " << endl;
  cp2 = cp;
  dp2 = dp;
  cp2--; cp2--;
  dp2++; dp2++;
  cout << "cp  - cp2 = " << (cp - cp2) << endl;
  cout << "dp2 - dp  = " << (dp2 - dp) << endl;
  cout << "(int)cp  - (int)cp2 = " << ((int)cp - (int)cp2) << endl;
  cout << "(int)dp2 - (int)dp  = " << ((int)dp2 - (int)dp) << endl;
  Attesa("terminare");
  return 0;
}
void Attesa(char * str) {
  cout << "\n\n\tPremere return per " << str;
  cin.get();
}
/*
 * L'aritmetica dei puntatori prende sempre in considerazione la dimensione del
 * puntatore.
 * Un puntatore e', in realta', un tipo peculiare di dato predefinito che ha una
 * propria aritmentica.
 */
```

Funzione main

Ad un programma è possibile fornire, al momento della sua esecuzione, dei parametri iniziali che possono modificare l'esecuzione e rendere quindi la scrittura del programma più generica.

Per esempio il nome del file da trattare invece di essere fissato all'interno del codice può essere passato al momento dell'avvio del programma.

In C vengono passate ad ogni programma due variabili normalmente chiamate "argc" ed "argv", che consentono appunto questo passaggio di parametri.

argc indica il numero di stringhe passate

argv è un vettore di puntatori a caratteri, stringhe, che contiene tutti i parametri forniti al momento dell'avvio.

argv[0] contiene sempre il puntatore alla stringa che rappresenta il nome con cui il programma è stato lanciato.

argc è quindi sempre almeno uguale ad uno in quanto è sempre presente in argv[0] il nome del programma.

L'ultimo puntatore significativo di argv è argv[argc-1].

argv[argc] è un puntatore NULL ed è l'ultimo elemento del vettore lagalmente accessibile.

Esempio:

cc -v -c source.c

argc	4
argv[0]	"cc"
argv[1]	"-v"
argv[2]	"-c"
argv[3]	"source.c"
argv[4]	NULL

Ritorna

```
/* ARG1.CPP Illustra le variabili usate per accedere
 * agli argomenti di un comando
 *      argc            argv
 */
#pragma hdrstop
#include <condefs.h>
#include <iostream.h>

//---------------------------------------------------------------------------
#pragma argsused
void Attesa(char *);
int main(int argc, char* argv[]) {
  int count;

  /* Visualizza gli argomenti del comando */
  cout << endl << "Argomenti del comando:" << endl;
  for( count = 0; count < argc; count++ )
     cout << "  argv[" << count << "]   " << argv[count] << endl;

  Attesa("terminare");
  return 0;
}
void Attesa(char * str) {
  cout << "\n\n\tPremere return per " << str;
  cin.get();
}
```

Ritorna

```
/* Arg2.cpp
 * Argomenti del main, comprese le variabili di ambiente
 */
#pragma hdrstop
#include <condefs.h>
#include <iostream.h>

//---------------------------------------------------------------------------
#pragma argsused
void Attesa(char *);
int main(int argc, char* argv[], char *env[]) {
  int i;

  cout << "Il valore di argc è " << argc << endl << endl;
  cout << "Sono stati passati " << argc << " argomenti al main"
       << endl << endl;

  for (i = 0; i < argc; i++)
     cout << "   argv[" << i << "]: " << argv[i] << endl;

  cout << endl << "Le stringhe di ambiente del sistema sono:" << endl << endl;

  for (i = 0; env[i] != NULL; i++)
     cout << "   env[" << i << "]: " << env[i] << endl;
  Attesa("terminare");
  return 0;
}
void Attesa(char * str) {
  cout << "\n\n\tPremere return per " << str;
  cin.get();
}
```

Conversione di puntatori

Non è ammesso convertire puntatori qualificati (const, volatile) in puntatori non qualificati.

Un puntatore a void può puntare a qualsiasi cosa.

Fatta eccezione di void * è illegale assegnare ad un puntatore di un tipo un puntatore di un altro tipo senza utilizzare il cast.

I puntatori possono essere convertiti mediante l'operatore (cast) in puntatori ad altre entità.

Un puntatore ad un oggetto può essere convertito in puntatore ad un tipo diverso, ma può essere utilizzato per accedere effettivamente alla memoria solo se i requisiti di allineamento del nuovo tipo sono uguali o meno stringenti di quelli del tipo di partenza.

Lo standard impone solo che il tipo con minori requisiti di allineamento sia char.

Esempio.

```
int i = 5,   *pi,   *pii;
float f = 3.7, *pf;
pi = &i;
pf = (float *)pi;            Operazione lecita ma ATTENZIONE!!
      *pf non punta ad un elemento valido!
pii = (int *)pf;             *pii punta ad un elemento VALIDO
      perché *pf punta nella stessa locazione di *pi
```

Un puntatore void rappresenta un puntatore a qualunque tipo di dato, l'uso di void * garantisce che la dimensione del puntatore sia almeno uguale alla dimensione del più grande puntatore implementato e quindi possa contenere qualunque puntatore.

Qualsiasi puntatore può essere trasformato in void * e poi riportato al suo tipo originale senza alcuna perdita di informazione.

Esempio:

```
int *pi, *pii;
void *p;
p = (void *)pi;
pii = (int *) p;
```

Tra puntatori e interi l'unica conversione ammessa è da un puntatore in un tipo intero, per ogni implementazione esiste almeno un tipo intero (short, int o long) adatto, ma non è necessariamente lo stesso per ogni implementazione.

Ritorna

```
/* Passpunt.cpp
 * Esempio di conversione tra puntatori
 */
#pragma hdrstop
#include <condefs.h>
#include <iostream.h>
#include <conio.h>
//---------------------------------------------------------------------------
#pragma argsused
int i = 5;
float f = 1.5;
int *pi, *pii;
float *pf;
double *pd;
void *p;
void Attesa(char *);
int main(int argc, char* argv[]) {
  cout << "i = " << i << endl;
  cout << "f = " << f << endl;
  pi = &i;
  pf = &f;
  cout << "pi   = &i = " << pi << endl;
  cout << "pf   = &f = " << pf << endl;
  cout << "*pi  = " << *pi << endl;
  cout << "*pf  = " << *pf << endl;
  p = (void *)pi;
  cout << "p    = (void *)pi = " << p << endl;
  pii = (int *)p;
  cout << "pii  = (int *)p   = " << pii << endl;
  cout << "*pii = " << *pii << endl;
  Attesa("continuare");

  p = (void *)pf;
  cout << "p    = (void *)pf  = " << p << endl;
  pd = (double *)p;
  cout << "pd   = (double *)p = " << pd << endl;
  cout << endl << "Attenzione!!" << endl
       << "Il puntatore pd punta nella stessa locazione di pf"  << endl
       << "ma si aspetta di trovare un elemento double" << endl << endl;
  cout << "*pd  = " << *pd << endl;
  pi = (int *)pf;
  cout << "pi   = (int *)pf = " << pi << endl;
  cout << endl << "Attenzione!!" << endl
       << "Il puntatore pi punta nella stessa locazione di pf" << endl
       << "ma si aspetta di trovare un elemento int" << endl   << endl;
  cout << "*pi  = " << *pi << endl;
  Attesa("continuare");

  pi = &i;
  cout << "pi   = &i          = " << pi << endl;
  pf = (float *)pi;
  cout << "pf   = (float *)pi = " << pf << endl;
  cout << endl << "Attenzione!!" << endl
       << "Il puntatore pf punta nella stessa locazione di pi" << endl
       << "ma si aspetta di trovare un elemento float" << endl << endl;
  cout << "*pf  = " << *pf << endl;
  pii = (int *)pf;
  cout << endl << "Mentre in questo caso i tipi corrispondono" << endl;
  cout << "pii  = (int *)pf = " << pii << endl << endl;
  cout << "*pii = " << *pii << endl;
  Attesa("terminare");
  return 0;
}
void Attesa(char * str) {
  cout << "\n\n\tPremere return per " << str;
  cin.get();
  clrscr();
}
```

Ritorna

```
/* Printv.cpp
 * Funzione di stampa che usa i puntatori void
 */
#pragma hdrstop
#include <condefs.h>
#include <iostream.h>

//---------------------------------------------------------------------------
#pragma argsused
enum tipo { reale, carattere, intero };
void Print(void *, tipo);
void Attesa(char *);
int main(int argc, char* argv[]) {
  int i   = 47;
  float f = 6.28;
  char c  = 'z';

  Print(&i, intero);
  Print(&f, reale);
  Print(&c, carattere);
  Attesa("terminare");
  return 0;
}
void Print(void *number, tipo type) {
  switch (type)         {
     case reale:
                cout << "float:     " << *((float *)number) << endl;
                break;
     case carattere:
                cout << "carattere: " << *((char *)number)  << endl;
                break;
     case intero:
                cout << "intero:    " << *((int *)number)   << endl;
                break;
  }
}
void Attesa(char * str) {
  cout << "\n\n\tPremere return per " << str;
  cin.get();
}
/*
 * Il concetto di puntatore come "una variabile contenente l'indirizzo di
 * un'altra variabile", offre interessanti possibilita'.
 * Se si conosce sempre la dimensione di un puntatore, si puo' passare a una
 * funzione l'indirizzo di un qualunque tipo di dato, la funzione puo'
 * effettuare una conversione di tipo sull'indirizzo, per ottenere un puntatore
 * del tipo corretto (basandosi su altre informazioni).
 * Un puntatore void rappresenta un puntatore a qualunque tipo di dato.
 * L'uso di void garantisce che la dimensione del puntatore sia almeno uguale
 * alla dimensione del piu' grande puntatore implementato
 */
```

Strutture

Una struttura è un raggruppamento di più variabili, non necessariamente dello stesso tipo, che formano un'unica entità, risulta una collezione non ordinata di oggetti distinti mediante il nome.

Sintassi:

```
struct [identificatore] {lista-campi};
```

Esempio:

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

L'operatore binario di accesso ai campi è il punto '.'

Esempio:

```
struct data oggi;
n = oggi.mese;
```

Ogni struttura introduce un nuovo ambiente nel quale è consentito ridefinire gli identificatori.

Se si omette il nome della struttura il tipo è dichiarato senza nome e non più utilizzabile in seguito.

Strutture definite con definizioni differenti, ma identiche campo per campo non sono considerate dello stesso tipo, la compatibilità è nominale.

Strutture e vettori sono costruttori combinabili per ottenere gerarchie più complesse.

E' possibile impiegare puntatori per scandire vettori di strutture

Sia struct data *pt;

le scritture (*pt).giorno; e pt->giorno sono equivalenti.

E' ammesso impiegare il nome del tipo di una struttura all'interno della sua stessa dichiarazione per dichiarare alcuni campi (dichiarazione ricorsiva).

Esempio:

```
struct nodo {
        int key;
        struct nodo *next;
}
```

Operazioni sulle strutture

selezione campi operatori . ed ->
ricavarne l'indirizzo operatore &
assegnamento operatore =
passarle come argomento di una funzione (viene copiata)
produrle come risultato di una funzione

Il punto permette di leggere un determinato membro contenuto in una struttura.

La freccia permette la lettura di un membro attraverso un puntatore alla struttura.

Entrambi gli operatori, punto e freccia, godono del massimo livello di precedenza, insieme a () ed []

Semantica dell'operatore di assegnamento:

sia struct data oggi, domani;

scrivere:

```
domani = oggi;
```

equivale

```
domani.giorno = oggi.giorno;
domani.mese   = oggi.mese;
domani.anno   = oggi.anno;
```

Sovrapposizione di variabili “union”

Un’unione è una variabile che ha la capacità di contenere uno e solo uno fra tanti differenti tipi di oggetti forniti di nome in un certo istante di tempo, indipendentemente dai tipi di questi oggetti
Lo scopo di una unione è il riutilizzo di una locazione di memoria o variabile.
Le zone di memoria per tutti i membri dell’unione iniziano allo stesso indirizzo
sizeof(unione) è quello del membro più grande
Il valore di uno solo dei membri dell’unione può essere memorizzato all’interno dell’unione in un dato momento.
Il costrutto ‘union’ alloca abbastanza memoria per contenere uno alla volta i campi indicati, quindi viene allocata la memoria necessaria per il campo di dimensione maggiore
Esempio:

```
struct elem {
        int tipo;
        union {
                char    c_val;
                int     i_val;
                double  d_val;
        } valore;
} tabella[100];
```

Per accedere ai campi avremo: tabella[3].valore.i_val
E’ responsabilità del programmatore accedere al solo campo significativo di una union, non esiste modo di sapere quale sia l’ultimo campo assegnato.
In pratica si utilizzano informazioni in altri campi o si ricavano dal contesto.
L’uso della union è consigliato quando si utilizzano grandi strutture dati in tempi diversi per aspetti diversi.
Per passare ad una funzione argomenti di tipo diverso si possono utilizzare i campi di una union e passare alla funzione l’union stessa.
Il contenuto di una union va riletto usando lo stesso tipo di variabile impiegato per scriverlo
Possono servire per risparmiare spazio quando si devono memorizzare diversi tipi di dati in alternativa l’uno all’altro, il loro uso più comune consiste nel rappresentare caselle speciali che devono essere interpretate in un dato modo (buffer di comunicazione con i device)
Esempio particolare:

```
union {
        int i;
        struct {
                unsigned char a;
                unsigned char b;
        }c;
}u;
```

Questa struttura consente di esaminare il formato intero degli int in memoria.

```
u.i = 0x1234;
printf(“%2x,%2x\n”, u.c.a, u.c.b);
```

otteniamo il risultato 34,12 i due byte sono scambiati; i micro processori Intel usano come codice interno il codice binario ma con i byte scambiati, sono memorizzati da quello meno significativo)

Definizione di tipo typedef

Sintassi: typedef tipo dichiaraz-strutt;
Formalmente typedef è una classe di memoria ma il significato è diverso: gli identificatori dichiarati vengono considerati equivalenti al tipo dato.
Con typedef non si dichiarano tipi nuovi ma si danno nomi a tipi esistenti o dichiarati nel typedef stesso.
L'uso di typedef permette di cambiare tipo ad un'intera classe di oggetti sostituendo la definizione in un unico punto del programma.
Typedef permette una certa semplicità di scrittura ed una migliore comprensibilità dei programmi.
Esempio:

```
typedef int (*punt)(void);
```

Dichiara 'punt' come puntatore a funzione senza argomenti che produce valori interi
se scriviamo

```
punt ptr, vett[8];
```

definiamo una variabile 'ptr' e un vettore di otto elementi 'vett' di tipo puntatore a funzioni a valori interi.

Note

Non si possono confrontare strutture o unioni per nome, il solo modo per eseguire la comparazione consiste nel verificare individualmente ogni membro della struttura.
La dimensione di una struttura non è soltanto la somma della dimensione dei suoi campi ma a seconda della macchina utilizzata la dimensione deve tenere conto degli allineamenti sulle parole macchina.
Le restrizioni dovute agli allineamenti potrebbero creare dei buchi cambiando la disposizione fisica della struttura, quindi non è portabile l'utilizzo di costanti legate all'hardware per rappresentare gli offset dei membri di una struttura.
Un metodo per forzare gli allineamenti consiste nell'utilizzare le unioni
Esempio:

```
union device_data {
        int dummy;  /* Per l'allineamento alla word */
        char inf[6];
};
```

Questo può essere utile nella chiamata ad una risorsa di sistema operativo, un device drive o una specifica periferica hardware che potrebbe aver bisogno che i dati trasmessi o ricevuti abbiano un allineamento ben specifico.
Ovviamente ci si aspetta che si prenda dall'unione solo quello che si è precedentemente messo, il compilatore non fa nessun controllo.
L'inizializzazione delle unioni deve essere rappresentata come una espressione costante e può solamente inizializzare il primo membro dell'unione.
Per le strutture che contengono riferimenti a se stesse in generale è sempre possibile riferirsi ad un tipo incompleto con un puntatore, ma non è possibile usarlo come entità a se stante.
Esempio:

```
struct s1_tag {
        struct s2_tag *s2ptr;
        char s1data[100];
};
struct s2_tag {
        struct s1_tag *s1ptr;
        char s2data[100];
};
```

E' preferibile nel passaggio di strutture a funzioni passare un puntatore alla struttura piuttosto che la struttura stessa.

Non ritornare un puntatore ad una struttura che è di classe automatic

Esempio:

```
struct temp {
        int memb:
};
struct temp *func()
  {     struct temp tmp;
        .........
        return (&tmp);
}
  main()
  {      struct temp *pt;
        pt = func();
        a = pt->memb;  ???????? non funziona!
}
```

Per risolvere il problema è sufficiente cambiare la dichiarazione in

```
static struct temp tmp;
```

Il fatto che tmp sia o meno in scope è irrilevante, lo scope è relativo solamente agli identificatori e non ha nulla a che fare con le variabili e gli indirizzi.

Naturalmente è possibile ottenere l'indirizzo di un identificatore fino a quando esso rimane visibile, una volta che si ha l'indirizzo di un oggetto si può fare di quell'oggetto quello che si vuole (fino a quando esso esiste).

In C++ non è più necessario precedere la definizione di una variabile con la parola chiave "struct"

Esempio:

```
enum semaforo { verde, giallo, rosso };
struct incrocio {
        semaforo luce;
        int coda_auto;
};
incrocio incr2[12];             // La parola chiave "struct" può essere omessa
```

In C++ non è più necessario indicare il nome della "union"

Esempio:

```
struct record  {
        char *nome;
        char tipo;
        union {
                char *svalue;           // usato se type == 's'
                int     ivalue;         // usato se type == 'i '
        };
};
record myrecord;
cout << myrecord.ivalue << endl;            // Non è più necessario il nome della "union"
```

Ritorna

```
/* Structur.cpp
 * Esempio di struttura
 */
#pragma hdrstop
#include <condefs.h>
#include <iostream.h>

//---------------------------------------------------------------------------
#pragma argsused
struct animale {
  int peso;
  int nzampe;
};
void Attesa(char *);
int main(int argc, char* argv[]) {
  animale fido, fufi, pollo;    // Non è necessario indicare struct

  fido.peso = 15;
  fufi.peso = 37;
  pollo.peso = 3;

  fido.nzampe = 4;
  fufi.nzampe = 4;
  pollo.nzampe = 2;

  cout << "Il peso di fido  e'    " << fido.peso
       << " e ha " << fido.nzampe  << " zampe.\n";
  cout << "Il peso di fufi  e'    " << fufi.peso
       << " e ha " << fufi.nzampe  << " zampe.\n";
  cout << "Il peso di pollo e'     " << pollo.peso
       << " e ha " << pollo.nzampe  << " zampe.\n";
  Attesa("terminare");
  return 0;
}
void Attesa(char * str) {
  cout << "\n\n\tPremere return per " << str;
  cin.get();
}
```

Ritorna

```
/* Ptrstrc.cpp
 * Selezionare membri quando si ha un puntatore a una
 * struttura o a un oggetto
 */
#pragma hdrstop
#include <condefs.h>
#include <iostream.h>

//---------------------------------------------------------------------------
#pragma argsused
struct A {
  char  c;
  int   i;
  float f;
};
void Attesa(char *);
int main(int argc, char* argv[]) {
  A u;                          // Crea una struttura
  A *up   = &u;                 // Crea un puntatore alla struttura
  up->i   = 100;                // Seleziona un membro della struttura
  up->c   = 'f';
  (*up).f = 23.5;
  cout << u.c << endl
       << u.i << endl
       << u.f;
  Attesa("terminare");
  return 0;
}
void Attesa(char * str) {
  cout << "\n\n\tPremere return per " << str;
  cin.get();
}
```

Ritorna

```
/* Agregate.cpp
 * Inizializzazione di strutture
 */
#pragma hdrstop
#include <condefs.h>
#include <iostream.h>

//---------------------------------------------------------------------------
#pragma argsused
struct semplice {
  int i, j, k;
};
semplice A = { 100, 200, 300 };
struct {
// Il tipo della struttura non e' necessario poiché viene definita un'istanza
  int i;
  char *nome;
  float f;
} array_di_struct[] = {
     1, "primo",   1.1,
     2, "secondo", 2.2,
     3, "terzo",   3.3,
     4, "quarto",  4.4
};
void Funz(void);
void Attesa(char *);
int main(int argc, char* argv[]) {
  for(int p = 0; p < 4; p++) {
     cout << "array_di_struct[" << p << "].i = "
          << array_di_struct[p].i << endl;
     cout << "array_di_struct[" << p << "].nome = "
          << array_di_struct[p].nome << endl;
     cout << "array_di_struct[" << p << "].f = "
          << array_di_struct[p].f << endl;
  }
  Funz();
  Attesa("terminare");
  return 0;
}
void Funz(void) {
  semplice A[] = { {1, 2, 3}, {4, 5, 6}, {7, 8, 9} };
  cout << endl << "A[]" << endl;
  for(int i = 0; i < 3; i++)
     cout << A[i].i << ", " << A[i].j << ", " << A[i].k << endl;
  // Va bene anche per gli array static
  static semplice B[] = { 1, 2, 3, 4, 5, 6, 7, 8, 9 };
  cout << endl << "B[]" << endl;
  for(int i = 0; i < 3; i++)
     cout << B[i].i << ", " << B[i].j << ", " << B[i].k << endl;
}
void Attesa(char * str) {
  cout << "\n\n\tPremere return per " << str;
  cin.get();
}
/*
 * Il compilatore effettua l'inizializzazione di una variabile globale una sola
 * volta, al caricamento del programma; nel caso di inizializzazioni di
 * aggregati ogni volta che entra in un blocco, oppure ogni volta che una
 * funzione viene chiamata, un blocco di codice particolare inizializza le
 * variabili.
 * I valori di inizializzazione vengono copiati negli elementi della struttura
 * nell'ordine di apparizione.
 * L'inizializzazione aggregata per un array di oggetti con membri privati
 * ciascun elemento viene inizializzato chiamando il costruttore
 * Se si vuole cambiare il numero di elementi dell'array e' sufficiente
 * aggiungere o eliminare alcuni elementi dalla lista di inizializzazione
 */
```

Ritorna

```
/* Retstruc.cpp
 * Ritorna una struttura per modificare piu' di una variabile
 * nell'ambiente
 */
#pragma hdrstop
#include <condefs.h>
#include <iostream.h>

//---------------------------------------------------------------------------
#pragma argsused
struct X {
  int i;
  float j;
  long k;
};
X Funz(int, float, long);
void Attesa(char *);
int main(int argc, char* argv[]) {
  X a;
  a = Funz(99, 2.15, 3000);
  cout << "a.i = " << a.i << endl
       << "a.j = " << a.j << endl
       << "a.k = " << a.k << endl;
  Attesa("terminare");
  return 0;
}
X Funz(int ii, float jj, long kk) {
  X local;
  local.i = ii + 4;
  local.j = jj + 3.5;
  local.k = kk + 2000;
  return local; // Ritorna tutti i valori che fanno parte della struttura
}
void Attesa(char * str) {
  cout << "\n\n\tPremere return per " << str;
  cin.get();
}
/*
 * Se si racchiudono tutti i valori che si vogliono modificare in una struttura
 * che viene ritornata dalla funzione e' possibile modificare piu di un valore
 */
```

Ritorna

```
/* Unionex.cpp
 * Esempio di unione
 */
#pragma hdrstop
#include <condefs.h>
#include <iostream.h>

//---------------------------------------------------------------------------
#pragma argsused
struct aereo {
  int apert_ali;
  int passeggeri;
  union {                 // Unione anonima
     float carburante;
     int   nbombe;
     int   contenitori;
  };
} caccia, bombardiere, trasporto;
void Attesa(char *);
int main(int argc, char* argv[]) {
  caccia.apert_ali   = 40;
  caccia.passeggeri  = 1;
  caccia.carburante  = 12000.0;

  bombardiere.apert_ali   = 90;
  bombardiere.passeggeri  = 12;
  bombardiere.nbombe      = 40;

  trasporto.apert_ali   = 106;
  trasporto.passeggeri  = 4;
  trasporto.contenitori = 42;

  trasporto.carburante   = 18000.0;
  caccia.contenitori     = 4;

  cout << "Il caccia trasporta      " << caccia.nbombe      << " bombe\n";
  cout << "Il bombardiere trasporta " << bombardiere.nbombe << " bombe\n";
  Attesa("terminare");
  return 0;
}
void Attesa(char * str) {
  cout << "\n\n\tPremere return per " << str;
  cin.get();
}
```

Organizzazione della memoria

heap

Si trova alla sommità dello spazio di memoria del programma ed è la sorgente per l'allocazione della memoria dinamica

stack

Contiene i dati in elaborazione

data

Contiene i dati statici e quelli esterni variabili

code

Contiene le istruzioni macchina.

La memoria per le variabili statiche ed esterne è allocata e inizializzata solo una volta al momento del caricamento dello stesso nella RAM

La memoria per le variabili auto o register è allocata ogni volta che si accede alla funzione nella quale sono impiegate e rilasciate quando il flusso di esecuzione lascia la funzione.

L'operatore sizeof determina il numero di byte allocati per la variabile indicata.

La memoria può essere allocata in modo dinamico utilizzando le funzioni di libreria stdlib.h

La funzione malloc accetta come argomento il numero di byte di memoria richiesti per essere allocati dinamicamente e ritorna un puntatore ad un blocco di memoria libera prelevata dalla heap

La funzione free trasferisce la memoria allocata dinamicamente all'insieme di memoria della heap rendendola disponibile per altre allocazioni.

L'utilizzo della funzione free diventa essenziale se l'area di memoria è allocata all'interno di una funzione in quanto ad ogni chiamata viene allocata nuova memoria.

Ritorna

```
/* SumNum.cpp
 * Eseque la somma di un numero variabile di numeri interi
 * Utilizza un array di lunghezza variabile
 */
#pragma hdrstop
#include <condefs.h>
#include <iostream.h>
#include <iomanip.h>
//---------------------------------------------------------------------------
#pragma argsused
void Attesa(char *);
int main(int argc, char* argv[]) {
  long *nums;            /* Puntatore ad un array di numeri */
  short quanti;          /* Numero di interi da considerare */
  short inum;            /* Indice dell' array */
  long sum;              /* Totale della somma */

  cout << "Quanti sono i numeri da sommare ?  ";
  cin  >> quanti;
  if(quanti < 2) {
    cout << "Input non accettabile" << endl;
    Attesa("terminare");
    exit(1);
  }
  /* Alloca la memoria dinamica per l'array */
  nums = new long[quanti];
  if(nums == NULL) {
    cout << "Manca memoria" << endl;
    Attesa("terminare");               getchar();
    exit(2);
  }
  for(inum = 0; inum < quanti; ++inum) {
    cout << "Introdurre #" << setw(3) << inum+1 << ": ";
    cin  >> *(nums+inum);
  }
  for (sum = inum = 0; inum < quanti; ++inum) {
    sum += *(nums+inum);
    cout << setw(3)  << inum+1 << ":"
         << setw(5)  << nums[inum]
         << setw(10) << sum << endl;
  }
  cout << endl << "La somma dei "  << quanti
       << " numeri introdotti e' " << sum << endl;
  delete [] nums;
  Attesa("terminare");
  return 0;
}
void Attesa(char * str) {
  cout << "\n\n\tPremere return per " << str;
  cin.ignore(4, '\n');
  cin.get();
}
```

File

Un file è un contenitore di dati a cui viene assegnato un nome.

In C è consentito operare solamente sui file costituiti da sequenze lineari di byte.

La prima operazione da eseguire prima di scrivere o leggere un file è di aprirlo

```
FILE *fp;
fp = fopen("nome_file", "modo");
```

Associa al puntatore 'fp' il file indicato.

La definizione del tipo FILE è contenuta nel file stdio.h

Le modalità di apertura di un file sono:

r	sola lettura	se non esiste restituisce NULL
w	sola scrittura	se esiste verrà riscritto se non esiste verrà creato
r+	lettura e scrittura	se non esiste restituisce NULL
w+	lettura e scrittura	se esiste verrà riscritto se non esiste verrà creato
a	scrittura a fine file	se esiste verrà aggiunto il nuovo contenuto alla fine del file se non esiste verrà creato
a+	lettura e scrittura a fine file	come a ma può anche essere letto
r+b	lettura e scrittura binaria	
w+b	scrittura e lettura binaria	

Usando il formato binario i byte sono letti esattamente come si trovano nel file senza conversioni.

Il vantaggio dei file binari è la registrazione dei numeri in formato intero

Dopo aver terminato le operazioni di lettura e scrittura bisogna eseguire l'operazione di chiusura

```
fclose(fp);
```

Il sistema prevede dei file predefiniti

stdin	standard input, tastiera
stdout	standard output, video
stderr	standard error, video

Lettura e scrittura

n = fread(buf, dim, elem, fp);

char buf[..]; vettore dove devono essere trasferite le informazioni lette

int dim; dimensione in byte di un elemento del vettore

int elem; numero di elementi da leggere

FILE *fp; puntatore al file;

fread restituisce il numero di elementi letti dal file

- Se restituisce un numero negativo significa che c'è stato un errore

- Se restituisce zero significa che è stata raggiunta la fine del file

Le operazioni di lettura avvengono in modo sequenziale e mantengono traccia nel punto in cui si è arrivati nella lettura

n = fwrite(buf, dim, elem, fp);

fwrite restituisce il numero di elementi scritti

- Se restituisce un numero negativo c'è stato un errore

- Se restituisce un numero minore di elem il file ha raggiunto la dimensione massima ammessa dal sistema

Posizionamento del puntatore

La funzione fseek consente di muovere il puntatore di lettura/scrittura in una qualunque posizione del file

err = fseek(fp, n, mod);

FILE *fp; puntatore al file

long n: numero di byte dello spostamento (se negativo viene spostato indietro)

int mod; posizione di partenza

- 0 inizio del file
- 1 posizione corrente
- 2 fine file

- Se fseek restituisce un numero diverso da zero significa che c'è stato un errore.

n = ftell(fp);

Restituisce la posizione corrente del puntatore

rewind(fp); riposiziona il puntatore all'inizio del file = fseek(fp, 0L, SEEK_SET);

Lettura e scrittura formattata

fprintf(); scrive sul file

fscanf(); legge da file

La formattazione è la stessa delle funzioni printf()e scanf()

Lettura e scrittura di righe di caratteri

s = fgets(buf, n, fp);

fgets legge una riga da un file, intendendo una riga la sequenza di caratteri terminata da '\n'

char buf[..]; vettore dove viene trasferita la riga letta

int n; numero di char da leggere

FILE *fp; puntatore al file

char *s; puntatore coincidente all'indirizzo di buf se la funzione è andata a buon fine altrimenti vale NULL

fputs(buf, fp);

Scrive una riga in un file, buf deve contenere una stringa terminata con '\0'

Lettura e scrittura di caratteri

c = fgetc(fp); legge un carattere dal file puntato da fp

int c; carattere letto dal file oppure EOF se il puntatore è posizionato alla fine del file.

EOF è una costante definita in stdio,h

fputc(c, fp); scrive il carattere c nel file

n = feof(fp); restituisce 1 se il puntatore è posizionato alla fine del file

fflush(fp); scarica il contenuto del buffer, associato al puntatore, su disco

Il buffer è gestito dal sistema operativo, le operazioni di scrittura non sono eseguite su disco ogni volta ma il sistema operativo ottimizza gli accessi al disco.

Secondo lo standard ANSI

- Non si può leggere un singolo carattere da tastiera senza aspettare il return
- Non si possono leggere i tasti funzione e i tasti cursore
- Non si può sapere se ci sono caratteri in attesa nel buffer della tastiera

Operazioni sui file

remove("nomefile");

Cancella il file, ritorna zero se è stato possibile cancellare il file

rename("vecchio", "nuovo");

Cambia il nome del file, ritorna zero se tutto è andato a buon file

Per non cancellare accidentalmente un file si può creare un file senza nome

FILE *tf;

tf = tmpfile();

Viene automaticamente cancellato quando viene chiuso

Ritorna

```
/* File.cpp
 * Scrive N coppie di numeri interi sul file "file_1"
 * chiude il file_1, lo riapre in lettura
 * legge le coppie di numeri, esegue la somma
 * e scrive il risultato sul "file_2"
 */
#pragma hdrstop
#include <condefs.h>
#include <stdio.h>
#include <stdlib.h>
//---------------------------------------------------------------------------
#pragma argsused
int main(int argc, char* argv[]) {
  FILE *idf1, *idf2;
  int n, i, num1, num2, som;

  printf("Quante coppie di numeri devo leggere? > ");
  scanf("%d", &n);
  idf1 = fopen("file_1", "w");
  if (idf1 == NULL) {
     printf("Non posso scrivere su file_1\n");
     exit(0);
  }
  for (i = 1; i <= n; i++) {
     printf("%3d coppia\n", i);
     printf("   primo numero  "); scanf("%d", &num1);
     fprintf(idf1, "%4d", num1);
     printf(" secondo numero  "); scanf("%d", &num2);
     fprintf(idf1, "%4d", num2);
  }
  fclose(idf1);
  idf1 = fopen("file_1", "r");
  if (idf1 == NULL) {
     printf("Non posso leggere il file_1\n");
     exit(0);
  }
  idf2 = fopen("file_2", "w");
  if (idf1 == NULL) {
     printf("Non posso scrivere su file_2\n");
     exit(0);
  }
  for (i = 1; i <= n; i++) {
     fscanf(idf1, "%d%d", &num1, &num2);
     som = num1 + num2;
     fprintf(idf2, "%6d", som);
  }
  fclose(idf1);
  fclose(idf2);
  while(getchar() != '\n');
  getchar();
  return 0;
}
```

Preprocessore

Le linee del file sorgente che iniziano con il carattere # sono direttive per il compilatore C
Una direttiva termina con il carattere newline, se la si vuole far proseguire nella linea successiva il newline va preceduto da '\'
Possono servire per realizzare:

Inclusione di file: #include viene inserito il testo del file indicato, sia esso un file di libreria o altro file che costituisce il programma

Macro espansione: una stringa del testo può essere sostituita ogni volta che è presente nel testo con un'altra stringa definita in precedenza. Per la definizione di valori numerici fissi è meglio usare le costanti evitando l'uso del preprocessore.

Compilazione condizionale: una parte del codice sorgente può essere compilata solo in determinati casi, cioè se è stato definito un dato parametro tramite la direttiva #define.

Controllo del compilatore: per impostare le modalità di compilazione, e ottimizzazioni

Inclusione di file ausiliari

Significa sostituire testualmente la linea che contiene la direttiva con il contenuto del file indicato.
Sintassi:

```
#include <nome-file>     oppure
#include "nome-file"
```

Nel primo caso il file viene ricercato direttamente nelle directory standard delle librerie
E' una convenzione di terminare il nome dei file da includere con ".h"
L'utilizzo più tipico è di localizzare in un unico file le dichiarazioni di macro e di variabili comuni a più sorgenti di un unico programma.
Un file di inclusione può a sua volta contenere la direttiva di inclusione.

Definizione di macro

Significa la sostituzione testuale di una stringa con un'altra.
Sintassi:

```
#define identificatore stringa
```

Tutte le volte che "identificatore" è incontrato nel sorgente è sostituito con "stringa"
Un utilizzo comune è la definizione di costanti con nomi significativi
Sintassi di una macro:

```
#define identificatore(param1, param2,...,param-n) stringa
```

Viene sostituito "identificatore" con "stringa" rimpiazzando i parametri formali con quelli attuali
Esempio:

```
#define abs(x) ((x) > 0 ? (x) : -(x))
```

scrivendo nel sorgente:

```
a = abs(rap);
```

Il preprocessore espanderà il codice in:

```
a = ((rap) > 0 ? (rap) : -(rap));
```

Nella scrittura di macro bisogna racchiudere la definizione e ogni parametro formale tra parentesi per evitare effetti indesiderati.
Esempio:

```
#define quadrato(x) (x*x)
```

Scrivendo: a = quadrato(b+c);
Verrà espanso: a = (b+c * b+c); che non è il quadrato della somma b+c

Se il nome di un parametro formale è preceduto da # verrà sostituito dal parametro attuale trasformato in stringa e racchiuso da doppi apici

Esempio:

```
#define esamina(x) (printf("Il valore di %s è %d\n", #x, x))
```

Scrivendo:

```
esamina(a+b);
```

Verrà espanso:

```
(printf("Il valore di %s è %d\n", "a+b2, a+b));
```

La sequenza ## all'interno di una definizione serve a far eliminare tutti gli spazi e caratteri equivalenti sia precedenti che seguenti alla sequenza stessa.

Esempio:

```
#define gen-nome(operaz, mezzo) operaz##_##mezzo
```

Scrivendo: gen_nome(scrivi, video)(x, y, carat);

Verrà espanso: scrivi_video(x, y, carat);

Esempio:

```
#define paste (front, back) front##back
```

se abbiamo paste(nome, 1)

otterremo la stringa "name1"

Nota:

Tra la fine del nome della macro e la parentesi tonda non è ammesso nessun carattere, nemmeno lo spazio

Esempio:

```
#define PRINT_N printf("n vale %d\n", n)
#define NOTA puts("Una linea può " \
        "continuare di sotto" \
        "con un backlash.")
```

Le macro possono espandere più istruzioni

Esempio:

```
#define scambia(x, y) \
    { \
        int i; \
        i = x; \
        x = y; \
        y = i; \
    }
```

Rispetto ad una funzione abbiamo una maggiore velocità di esecuzione ma rendono il codice più lungo

Esempio:

```
#define max(A, B)   ((A)  > (B)  ? (A)  : (B))
```

Viene espansa in codice in linea

se abbiamo `x = max(p+q, r+s);`

viene sostituita con `x = ((p+q)  > (r+s)  ?  (p+q)  : (r+s));`

La sostituzione può essere utilizzata per ogni tipo di dati.

Attenzione nel caso di

```
max(i++, j++)
```

i ed j verranno incrementati due volte difatti la sostituzione risulta:

```
(i++) > (j++) ? (i++)  : (j++)
```

E' possibile cancellare una macro precedente con

```
#undef identificatore
```

Compilazioni condizionali

Permettono di far compilare o meno certe porzioni di codice

1) #if espr-costante

Se l'espressione è vera il codice fino alla direttiva #else oppure #elsif oppure #endif viene copiato nel testo da compilare, se è falso viene eliminato

E' ammesso l'operatore defined identificatore

2) #ifdef identificatore

L'espansione è condizionata dal fatto che identificatore sia stato definito con un #define

#ifndef identificatore

L'espansione è condizionata dal fatto che identificatore non sia stato definito con un #define

#else

Specifica una alternativa ad una precedente direttiva condizionale

#elsif espr-costante

Specifica una condizione alternativa da valutare se la condizione precedente è falsa.

#endif

Deve chiudere ogni dichiarazione di compilazione condizionale

Esempio:

```
#ifdef  TURBOC
#define Jtextcolor textcolor
#else
#define Jtextcolor(col) ((void)_settextcolor(col))
#endif
```

Direttive per il compilatore

1) #error messaggio

Provoca l'emissione del "messaggio con la terminazione in errore della compilazione

Esempio:

```
#if !defined/SYSTEM)
#error     SYSTEM deve essere definito
#endif
```

2) #pragma direttive (dipende dal compilatore)

Permette di variare opzioni di compilazione come informazioni per il debug, tipo di ottimizzazione

3) #line costante-intera nome-file

Fa credere al compilatore che la linea seguente sia il numero "costante-intera" del file "nome-file"

Ad esempio alla fine di una inclusione è importante far sapere al compilatore che ha ripreso il sorgente originario perché possa indicare il numero corretto della linea dove può eventualmente aver trovato un errore.

Ritorna

```
/* PMacro1.cpp
 * Utilizzare il precompilatore cpp32
 */
#pragma hdrstop
#include <condefs.h>
#include <iostream.h>
#include "PMacro1.h"

#pragma argsused
void Attesa(char *);
int main(int argc, char* argv[]) {
  int n = -5;
  char buf[M];

  n = abs(n + 3);
  Attesa("terminare");
  return 0;
}
void Attesa(char * str) {
  cout << "\n\n\tPremere return per " << str;
  cin.get();
}


//---------------------------------------------------------------------------
/* Pmacro1.h
 */
//---------------------------------------------------------------------------
#ifndef Macro1H
#define Macro1H
#define M 30
#define abs(x) ((x) > 0 ? (x) : -(x))
//---------------------------------------------------------------------------
#endif
```

Funzioni inline

Una funzione definita per mezzo di un macro del preprocessore, consente di abbreviare il codice, aumentare la leggibilità, ridurre gli errori ed eliminare l'overhead di una chiamata di funzione.
Esempio:

```
// Macro per visualizzare una stringa e un valore
#define print(str, var) cout << str " = " << var << endl
```

Le funzioni definite come macro hanno lo svantaggio di non essere delle vere funzioni e non viene effettuato il controllo statico di tipo.
Per funzioni di piccole dimensioni per una maggiore velocità si potrebbe preferire l'uso di macro del preprocessore, al posto delle funzioni, per evitare l'overhead della chiamata di funzione.
In C++ sono state introdotte le funzioni inline
Esempio:

```
inline int uno() { return 1; }
```

La parola chiave "inline" suggerisce al compilatore di espandere la funzione in linea come se fosse una macro. Invece di costruire il codice per la chiamata di funzione da mandare in esecuzione ogni volta che questa viene utilizzata, sostituisce nel punto in cui è presente la chiamata una copia del codice della funzione stessa.
Normalmente il compilatore, quando incontra la definizione di una funzione inline, genera il codice corrispondente e memorizza il punto di partenza di questo codice, quando poi incontra una chiamata di questa funzione inserisce una istruzione di salto per far eseguire le istruzioni presenti all'indirizzo della funzione.

Note:

Non si può dichiarare una funzione inline come una normale funzione.
Esempio: inline int one();
La parola chiave inline non ha effetto poiché non ha senso che una funzione sia inline se non gli viene fornito il codice da sostituire alla chiamata di funzione.
Le definizioni di funzioni inline devono apparire prima che le funzioni siano usate.
Inline rappresenta solo un suggerimento al compilatore che deciderà quale sia la soluzione migliore.
É possibile chiedere al compilatore di ignorare la parola chiave "inline"

Vantaggi

In una chiamata di funzione inline il compilatore controlla che l'uso dia corretto e poi sostituisce il codice della funzione alla chiamata, in questo modo, l'efficienza delle macro del preprocessore è combinata con il controllo di tipo delle normali funzioni.
Esempio:
Soluzione con una macro

```
#define MAX(a, b) (a > b ? a : b)
int x = 10, y = 11;
MAX(x, ++y);
        // Viene espansa in ( x > ++y ? x : ++y)
        // "y" viene incrementato due volte
```

Soluzione con una funzione inline

```
inline int max(int a, int b) {
        return ( a > b ? a : b );
}
max(x, ++y);
        // "y" viene incrementato una volta e poi assegnato a "b"
        // Il codice cella funzione max() viene espanso nel punto di chiamata
```

Il vantaggio di una funzione inline consiste nel fatto che permette di abbreviare la scrittura del codice, poiché la funzione è dichiarata e definita in un solo luogo; inoltre il codice è spesso più leggibile.
Se una funzione inline non viene mai chiamata, non viene generato alcun codice, per una funzione normale, il codice è sempre presente anche se la funzione non viene mai chiamata.
Le funzioni "inline"possono essere tracciate con un "souce level debugger"

Svantaggi

Aspetto negativo: modificando l'implementazione di una funzione inline occorrerà ricompilare le parti di codice che la chiamano mentre se è una funzione ordinaria non è necessario.
Le funzioni "inline" sono più affidabili delle macro ma non sempre possono sostituire elegantemente le macro.
Esempio:

```
double x = 12.5, y = 25.4;
int a = 23, b = 45;
long k = 456, z = 678;
double h = max(x, y);
long g = max(k, z);
int w = max(a, b)                    // Dovremmo avere tre versioni della funzione max()
```

Poiché una funzione inline duplica il codice per ogni chiamata di funzione, si potrebbe pensare che automaticamente aumenti lo spazio di memoria per il codice; per funzioni di piccole dimensioni ciò non è necessariamente vero.
Infatti una funzione richiede codice per passare gli argomenti, effettuare la chiamata e gestire il valore di ritorno; se la funzione inline ha dimensione minore della quantità di codice necessario per una normale chiamata di funzione si risparmia spazio di memoria.
I benefici di una maggiore velocità delle funzioni inline tendono a diminuire con l’aumento della dimensione della funzione; ad un certo punto l’overhead della chiamata di funzione diventa piccolo se confrontato con l’esecuzione del corpo della funzione.
Ha senso definire una funzione "inline" se è piccola ed è usata molte volte

Operatori sui bit

Il C nato come linguaggio per la programmazione di sistema, dovendo consentire la gestione di tutte le caratteristiche della macchina dispone di operatori che permettono di agire direttamente sulla rappresentazione interna dei valori.

Tali operatori agiscono solo su operandi interi e carattere e considerano la rappresentazione interna dei valori di tali tipi sia basata sulla aritmetica binaria.

Essi devono essere sempre utilizzati solo su entità considerate per la loro rappresentazione, mai per il loro valore, in caso contrario il programma non è portabile su altre macchine.

I valori prodotti da questi operatori dipendono fortemente dalla implementazione, dalla rappresentazione dei numeri che può essere:

in complemento a due (i numeri negativi sono x + 2^n)

in complemento a uno

in modulo e segno

Il loro uso è inevitabile per gestire certe caratteristiche dell'hardware.

Il loro uso corretto è di assegnare valori a singoli bit, modificarli ed esaminarli indipendentemente dal loro valore numerico.

~ Inverte i bit del suo argomento

Se gli interi sono rappresentati con 16 bit in complemento a due

x = 5	0000000000000101
~x = -6	1111111111111010

& Produce un bit a 1 se e solo se entrambi i bit degli operatori sono 1

x = 5	0000000000000101
y = 9	0000000000001001
x & y = 1	0000000000000001

| Produce un bit a 1 se e solo se almeno uno dei bit degli operandi è 1

x = 5	0000000000000101
y = 9	0000000000001001
x | y = 13	0000000000001101

^ (or esclusivo) Produce un bit a 1 se e solo se uno ma non entrambi i bit degli operandi è 1

x = 5	0000000000000101
y = 9	0000000000001001
x ^ y = 12	0000000000001100

Attenzione alla differenza tra l'operatore && ed &, così pure tra gli operatori || ed |; il primo è un operatore logico e può produrre solo un valore vero o falso(1 oppure 0) mentre il secondo può produrre qualsiasi valore.

Operatori di shift

<< I bit del primo operando sono spostati a sinistra di tanti posti quanti sono indicati dal secondo operando.

Esempio:

x = 5	0000000000000101	
x << 2	0000000000010100	x = 20

I bit che entrano da destra sono sempre zero

>> I bit del primo operando sono spostati a destra di tanti posti quanti sono indicati dal secondo operando.

Esempio:

x = 5	0000000000000101	
x >> 2	0000000000000001	x = 1

I bit che entrano da sinistra sono:

Se il primo operando è

non negativo entrano degli zeri

negativo dipende dalla implementazione, di solito il bit più significativo (bit di segno) rimane inalterato.

Esempio:

x = -5	1111111111111011		
x >> 1	0111111111111101	x = 32765	se entra un bit a zero
x >>1	1111111111111101	x = -3	se copia il bit di segno

Esempi particolari:

y = (x >> n) << n	Gli n bit di destra vengono persi con il primo shift e sostituiti da zeri con il secondo
y = x | (0x01 << n)	Pone ad 1 l'n-esimo bit di x (or bit a bit con il valore 1 spostato a sinistra nella posizione voluta.

Il risultato di uno shift è indefinito se il secondo operando è negativo o non minore del numero di bit utilizzati per rappresentare il primo operando.

Un unico scorrimento di tanti bit quanti sono quelli che compongono il primo operando produce un valore indefinito, se la stessa operazione viene effettuata in due passi il valore è sicuramente zero.

Ritorna

```
/* Bit.cpp
 * Operazioni sui bit
 */
#pragma hdrstop
#include <condefs.h>
#include <iostream.h>

//---------------------------------------------------------------------------
#pragma argsused
unsigned short a, b, c;
unsigned short mask1;
char ch;
int in, i;
long ln;
void Stampa(int, long);
void Attesa(char *);
int main(void) {
  cout << "sizeof(short) = " << sizeof(unsigned short) << endl;
  cout << "sizeof(int)   = " << sizeof(int) << endl;
  cout << "sizeof(long)  = " << sizeof(long) << endl;
  cout << "sizeof(char)  = " << sizeof(char) << endl;
  cout << "Introdurre il valore di a  ";
  cin  >> a;
  a = (unsigned short) a;
  cout << "Introdurre il valore di b  ";
  cin  >> b;
  b = (unsigned short) b;
  cout << endl << "a = " << a << "\t\t";   Stampa(sizeof(unsigned short), a);
  cout << endl << "b = " << b << "\t\t";   Stampa(sizeof(unsigned short), b);
  c = a & b;
  cout << endl << "a & b  = " << c << "\t"; Stampa(sizeof(unsigned short), c);
  c = a | b;
  cout << endl << "a | b  = " << c << "\t"; Stampa(sizeof(unsigned short), c);
  c = a ^ b;
  cout << endl << "a ^ b  = " << c << "\t"; Stampa(sizeof(unsigned short), c);
  c = ~a;
  cout << endl << "~a     = " << c << "\t"; Stampa(sizeof(unsigned short), c);
  c = ~a + 1;
  cout << endl << "~a + 1 = " << c << "\t"; Stampa(sizeof(unsigned short), c);
  Attesa("continuare");
  c = a << 2;
  cout << endl << "a << 2 = " << c << "\t";
  Stampa(sizeof(unsigned short), c);
  c = a >> 2;
  cout << endl << "a >> 2 = " << c << "\t";
  Stampa(sizeof(unsigned short), c);
  c = ~a >> 2;
  cout << endl << "~a >> 2 = " << c << "\t";
  Stampa(sizeof(unsigned short), c);
  cout << endl << endl << "Introdurre un carattere minuscolo: ";
  cin  >> ch;
  cout << endl << ch << "\t";       Stampa(sizeof(char), ch);
  ch = ch & ~32;
  cout << endl << ch << "\t";      Stampa(sizeof(char), ch);
  cout << endl << endl << "Introdurre un carattere maiuscolo: ";
  cin  >> ch;
  cout << endl << ch << "\t";      Stampa(sizeof(char), ch);
  ch = ch | 32;
  cout << endl << ch << "\t";      Stampa(sizeof(char), ch);
  Attesa("terminare");
  return 0;
}
void Stampa(int x, long n) {
  int i;

  for (i=x*8-1; i>=0; i--)
     cout << ((n >> (i)) & ~(~0<<1));
}
void Attesa(char * str) {
  cout << "\n\n\tPremere return per " << str;
  cin.ignore(4, '\n');
  cin.get();
}
```

I campi bit

In C è possibile suddividere un intero in campi di pochi bit di lunghezza

Sintassi:

[tipo-campo nome-campo* : costante-intera;

I campi bit vanno dichiarati all'interno di una struttura (struct, union)

Vengono riservati tanti bit quanto indicato in costante-intera.

Non si può applicare l'operatore & a tali campi.

Esempio:

```
struct elemento {
        char nome[32];
        unsigned int indirizzo;
        unsigned int tipo : 4;
        unsigned int classe : 3;
        unsigned int utilizzato : 1;
        unsigned int assegnato : 1;
} tabella [1000];
```

Si raccomanda di esplicitare sempre signed o usigned se il campo dichiarato è di tipo int, altrimenti il modo di memorizzazione dipende dalla implementazione.

Se un campo dovesse venire a trovarsi a cavallo di due parole macchina il compilatore può spostare il campo nella parola seguente lasciando alcuni bit inutilizzati.

Se si dichiarano campi senza nome esempio:

: 3;

si ottiene un campo di riempimento non utilizzato.

Se si dichiara un campo di lunghezza zero esempio:

unsigned int cp : 0;

si indica al compilatore che il prossimo campo deve iniziare allineato al prossimo int in memoria, lasciando se necessario un intervallo non utilizzato.

Un campo bit può essere solo di tipo int, unsigned int o signed int.

Campi adiacenti vengono raggruppati in un'area se sono disponibili i bit.

Membri diversi dai campi bit garantiscono che la memoria del successivo membro inizia sul confine allineato con int.

I campi bit possono essere usati in espressioni aritmetiche come gli altri interi

Non è possibile usare puntatori a campi bit ma solo puntatori alla zona indirizzabile che essi usano.

Ritorna

```
/* Strt_bit.cpp
 * Strutture con campi di bit
 */
#pragma hdrstop
#include <condefs.h>
#include <iostream.h>

//---------------------------------------------------------------------------
#pragma argsused
struct elemento {
   char nome[32];
   unsigned int indirizzo;
   unsigned int tipo: 4;
   unsigned int classe: 3;
   unsigned int utlizzato: 1;
   unsigned int assegnato: 1;
   unsigned int fine;
};

struct esempio {
   signed int riservato: 4;
   signed int velocita: 2;
   signed int reset:1;
   signed int flag:1;
};
void Attesa(char *);
int main(int argc, char* argv[]) {
   int dim;
   elemento elem;
   esempio ese[2], *pte;

   dim = sizeof(struct elemento);
   cout << "La dimensione della struttura elemento risulta: " << dim << endl;
   cout << "&elem            = " << &elem           << endl;
   cout << "&elem.nome       = " << &elem.nome      << endl;
   cout << "&elem.indirizzo  = " << &elem.indirizzo << endl;
   cout << "&elem.fine       = " << &elem.fine      << endl << endl;
   dim = sizeof(struct esempio);
   cout << "La dimensione della struttura esempio risulta: " << dim << endl;
   pte = &ese[0];
   cout << "&ese[0]  = " << pte << endl;
   pte = &ese[1];
   cout << "&ese[1]  = " << pte << endl;
   ese[0].flag = 1;
   if(ese[0].flag)
      cout << "flag impostato" << endl;
  Attesa("terminare");
  return 0;
}
void Attesa(char * str) {
  cout << "\n\n\tPremere return per " << str;
  cin.get();
}
```

Puntatori a funzioni

Una volta che una funzione è compilata e caricata per essere eseguita occupa un blocco di memoria che ha un certo indirizzo; di conseguenza è possibile assegnare ai puntatori l'indirizzo di partenza delle funzioni
Un puntatore a funzione risulta quindi una variabile che contiene l'indirizzo di una funzione.
Il nome della funzione è l'indirizzo della funzione stessa.
Esempio:

```
long Quadrato(int n) {
        return(long) n * n;
}
long (*pf)(int x);
```

Puntatore a funzioni che accettano un argomento di tipo int e ritornano un valore di tipo long

```
pf = Quadrato;          *pf   è la funzione puntata da pf.
q = (*pf)(a);           Assegna a q il quadrato di a, equivalente ad q = Quadrato(a);
```

Un puntatore ad una funzione può essere passato come argomento ad un'altra funzione:
Esempio:

```
long Disegna(int lato, long (*Figura)(int))  {
        return((*Figura)(lato));
}
```

La funzione Disegna può essere utilizzata nel modo seguente:

```
pf = Quadrato;
a = Disegna(5, pf);
```

Esempio

```
void (* funz_ptr)();
        funz_ptr è una variabile che punta ad una funzione senza argomenti ne valori di ritorno.
void *funz_ptr();
        funz_ptr è una funzione senza argomenti ne valori di ritorno
```

Array di puntatori a funzioni

Per selezionare una funzione si indicizza l'array e si dereferenzia il puntatore.
Si realizza il concetto di programma guidato da tabelle; invece di usare istruzioni condizionali si selezionano le funzioni da eseguire in base a una variabile di stato.
Esempio:

```
typedef double(*PF[3])(double, double);
PF arr = {Max, Min, Med};
double d = arr[1](7.5, 8.6);
        d assumerà il valore 7.5 in quanto è stata utilizzata la funzione Min
```

Ritorna

```
/* Funcpnt.cpp
 * Puntatori a funzione
 */
#pragma hdrstop
#include <condefs.h>
#include <iostream.h>

//---------------------------------------------------------------------------
#pragma argsused
void print(float);
void print_mess(float);
void print_float(float);

void (*fptr)(float);

void Attesa(char *);
int main(int argc, char* argv[]) {
  float pi = 3.14159;
  float due_pi = 2.0 * pi;

  print(pi);
  fptr = print;
  fptr(pi);             // Viene attivata print
  fptr = print_mess;
  fptr(due_pi);         // Viene attivata print_mess
  fptr(13.0);           // Viene attivata print_mess
  fptr = print_float;
  fptr(pi);             // Viene attivata print_float
  print_float(pi);
  Attesa("terminare");
  return 0;
}
void Attesa(char * str) {
  cout << "\n\n\tPremere return per " << str;
  cin.get();
}
void print(float dato) {
  cout << "Questa e' la funzione print." << endl;
}

void print_mess(float msg) {
  cout << "Il dato da visualizzare e' " << msg << endl;
}

void print_float(float ft) {
  cout << "Il dato da stampare e' " << ft << endl;
}
```

Ritorna

```
/* Functabl.cpp
 * Utilizzo di un array di puntatori a funzioni
 */
#pragma hdrstop
#include <condefs.h>
#include <iostream.h>

//---------------------------------------------------------------------------
#pragma argsused
// Macro per definire funzioni fittizie
#define DF(N) void N() { cout << "Funzione " #N " chiamata.." << endl; }

DF(A); DF(B); DF(C); DF(D); DF(E); DF(F); DF(G);

void (*funz_tab[]) () = { A, B, C, D, E, F, G };
void Attesa(char *);
int main(int argc, char* argv[]) {
   while(1) {
      cout << "Digitare un tasto da 'a' ad 'g' o 'q' per terminare" << endl;
      char c, c1;
      cin.get(c);
      cin.get(c1);               // Il secondo per CR
      if( c == 'q') break;       // Esce da while(1)
      if(c < 'a' || c > 'g') continue;
      (*funz_tab[c - 'a']) ();
   }
  Attesa("terminare");
  return 0;
}
void Attesa(char * str) {
  cout << "\n\n\tPremere return per " << str;
  cin.get();
}
/*
 * Con un array di puntatori a funzioni per selezionare una funzione
 * si indicizza l'array e si dereferenzia il puntatore.
 * Questo costrutto supporta il concetto di programma guidato da tabelle, si
 * si selezionano le funzioni da eseguire in base a una variabile di stato
 */
```

Ritorna

```
/* P_funz.cpp
 * Esempio di puntatori a funzione
 */
#pragma hdrstop
#include <condefs.h>
#include <iostream.h>

//---------------------------------------------------------------------------
#pragma argsused
int Somma(int, int);
int Sottraz(int, int);
void Calcolo(int, int, int (*)(int, int));
void Attesa(char *);
int main(int argc, char* argv[]) {
  int a, b, sc;

  cout << "Digitare un numero intero: ";
  cin  >> a;
  cout << "Digitare un altro numero intero: ";
  cin  >> b;
  cout << "Indicare l'operazione desiderata\n0 -- Somma\n1 -- Sottrazione\n";
  cin  >> sc;
  switch(sc) {
     case 0 : Calcolo(a, b, Somma);
              break;
     case 1 : Calcolo(a, b, Sottraz);
              break;
  }
  Attesa("terminare");
  return 0;
}
int Somma(int x, int y) {
  return (x + y);
}

int Sottraz(int x, int y) {
  return (x - y);
}

void Calcolo(int a, int b, int (*Oper)(int x, int y)) {
  int ris;

  ris = Oper(a, b);
  cout << "Risultato della operazione richiesta = " << ris << endl;
}
void Attesa(char * str) {
  cout << "\n\n\tPremere return per " << str;
  cin.ignore(4, '\n');
  cin.get();
}
```

Funzioni con un numero variabile di argomenti

Ad esempio printf può essere chiamata con un numero di argomenti variabile, essa è definita nel modo seguente:

```
void printf(const char *format, ...)
```

I tre punti indicano che al momento della chiamata verranno sostituiti da un numero arbitrario di argomenti di tipo ignoto, anche nessuno.

Per accedere agli argomenti si utilizzano le macro definite in stdarg.h

va_list

va_start

va_arg

va_end

Utilizzo delle macro:

- Dichiarare una variabile locale di tipo va_list

```
va_list arg_ptr;
```

- Inizializzarla

```
va_start(arg_ptr, s);
```

arg_ptr	variabile di tipo va_list
s	ultimo argomento dichiarato (almeno uno)

- Si accede al primo argomento non dichiarato (e ai successivi)

```
i = va_arg (arg_ptr, int);   tipo dell'argomento
```

- Prima di uscire eseguire

```
va_end (arg_ptr);
```

Se si desidera accedere nuovamente agli argomenti non dichiarati bisogna iniziare da capo con va_start

Tecniche per indicare il numero di argomenti passati

- dedotto dal contesto, valori di variabili esterne

- passare il numero come argomento in posizione fissa

- ricavarlo dall'esame di uno degli argomenti

- la fine degli argomenti è segnalata con un argomento con valore convenzionale (0, -1, NULL), funziona solo se gli argomenti sono tutti dello stesso tipo

E' responsabilità del programmatore non accedere oltre l'ultimo argomento effettivo.

Ritorna

```
/* F_arg.cpp
 * Utilizzo di funzioni con un numero variabile di argomenti
 */
#pragma hdrstop
#include <condefs.h>
#include <iostream.h>
#include <stdarg.h>
//---------------------------------------------------------------------------
#pragma argsused
int Max(int num_arg, ...);
void Attesa(char *);
int main(int argc, char* argv[]) {
  int a, b, c, d, e;
  int max;

  cout << "Indicare 5 numeri:  ";
  cin  >> a >> b >> c >> d >> e;
  max = Max(2, a, b);
  cout << "Il massimo tra "
       << a << ", " << b
       << " risulta " << max << endl;
  max = Max(3, a, b, c);
  cout << "Il massimo tra "
       << a << ", " << b << ", " << c
       << " risulta " << max << endl;
  max = Max(4, a, b, c, d);
  cout << "Il massimo tra "
       << a << ", " << b << ", " << c << ", " << d
       << " risulta " << max << endl;
  max = Max(5, a, b, c, d, e);
  cout << "Il massimo tra "
       << a << ", " << b << ", " << c << ", " << d << ", " << e
       << " risulta " << max << endl;
  Attesa("terminare");
  return 0;
}
int Max(int num_arg, ...) {
  va_list arg_ptr;
  int massimo, valore;

  va_start(arg_ptr, num_arg);
  massimo = va_arg(arg_ptr, int);
  while (--num_arg > 0)
     if ((valore = va_arg(arg_ptr, int)) > massimo)
        massimo = valore;
     va_end(arg_ptr);
  return massimo;
}
void Attesa(char * str) {
  cout << "\n\n\tPremere return per " << str;
  cin.ignore(4, '\n');
  cin.get();
}
```

Lettura di dichiarazioni complesse

int i; i è un intero
int *i; i è un puntatore ad un intero
int *i[3]; i è un vettore di tre puntatori ad interi
int (*i)[3]; i è un puntatore ad un vettore di tre interi
int *i(); i è una funzione che restituisce un puntatore ad un intero
int (*i)(); i è un puntatore ad una funzione che restituisce un intero.
int **i; i è un puntatore ad un puntatore ad un intero.
int *(*i)(); i è un puntatore ad una funzione che restituisce un puntatore ad un intero.
int *(*i[])(); i è un vettore di puntatori a funzioni che restituiscono puntatori a interi
int *(*(*i)())[10];
i è un puntatore ad una funzione che restituisce un puntatore ad un vettore di 10 puntatori ad un intero
int *(*i[5])[4]; i è un vettore di 5 puntatori ad un vettore di 4 puntatori ad un intero.
int i; i è un intero variabile, può contenere diversi valori
const int i; i è un intero costante, non gli si può assegnare nessun valoredopo la dichiarazione
int *i; i è un puntatore variabile ad un intero variabile
int * const i; i è un puntatore costante ad un intero variabile
const int *i; i è un puntatore variabile ad un intero costante
int const *i; i è un puntatore variabile ad un intero costante
const int * const i; i è un puntatore costante ad intero costante
const int *i[3]; i è un vettore di 3 puntatori ad interi costanti
int * const i[3]; i è un vettore di 3 puntatori costanti ad interi
const int * const i[3]; i è un vettore di tre puntatori costanti ad interi costanti
const int (*i)[3]; i è un puntatore ad un vettore di 3 interi costanti
int (* const i)[3]; i è un puntatore costante ad un vettore di 3 interi
const int (* const i)[3]; i è un puntatore costante ad un vettore di 3 interi costanti
const int *i(); i è una funzione che restituisce un puntatore ad un intero costante
int * const i(); i è una funzione costante che restituisce un puntatore ad un intero.
const int (*i)(); i è un puntatore ad una funzione che restituisce un intero costante
int (* const i)(); i è un puntatore costante ad una funzione che restituisce un intero
const int **i; i e' un puntatore ad un puntatore ad intero costante
int ** const i; i è un puntatore costante ad un puntatore ad un intero
int * const * i; i è un puntatore costante ad un puntatore ad un intero
int const **i; i è un puntatore ad un puntatore ad un intero costante
int const * const * i; i è un puntatore ad un puntatore costante ad un intero costante
const int * const * const i; i è un puntatore costante ad un puntatore costante ad un intero costante
int *(* const i[5])();

i è un vettore costante di 5 puntatori a funzioni che restituiscono puntatori ad int

int (* (* const (* const i[2])[4])())[6];

i è un vettore costante di 2 puntatori ad un vettore costante di 4 puntatori a funzioni che restituiscono puntatori ad un vettore di 6 interi

typedef double (*(*(*fp3)())[10])();
fp3 A;

fp3 è un puntatore ad una funzione senza argomenti che restituisce un puntatore ad un array di dieci puntatori a funzioni senza argomenti che restituiscono un double
A è una istanza di fp3

3° Modulo

- Astrazione
- Argomenti di default
- Scope resolution
- Riferimenti
- Overloading
    - Gestione dei nomi
- Classi
    - This
    - Definizioni
    - Membri static
    - Membri const
- Costruttori
    - Di default
    - Inizializzazione aggregata
    - Creazione dinamica
        - Array di oggetti
- Stream
    - Formattazione
    - Manipolatori
    - Funzioni membro
- Costruttori di copia
- Funzioni amiche
- Assegnamento
    - Alias
- Operatori
    - Puntatori intelligenti
- Conversioni

Introduzione al C++

Il C++ è stato sviluppato da Bjarne Stroustrup nel 1983.

Il C++ è essenzialmente una estensione del C standard con lo scopo di semplificare la gestione, la programmazione e la manutenzione dei sistemi software di grandi dimensioni.

I linguaggi di programmazione non devono fornire solo la possibilità di esprimere computazioni (trasformazioni I/O) ma anche di consentire di dare struttura alla descrizione e di organizzare in modo opportuno il processo produttivo.

Obiettivo: costruzione modulare e incrementale del software per estensione-specializzazione di componenti.

Nuovi concetti chiave:

> **Astrazione**: processo attraverso il quale si decide che una certa parte di informazione è dettaglio e si concentra l'attenzione su ciò che è considerato fondamentale per gli obiettivi da raggiungere.
>
> **Tipo di dato astratto**: connotato dalle operazioni che è possibile fare su quel dato, rende disponibile una serie di operazioni.
>
> **Ruolo della astrazione dei dati**: consente di concentrare l'attenzione sulle categorie concettuali dei dati del dominio applicativo, favorisce la modularità, la creazione di sottosistemi autonomi validati e promuove architetture cliente-servitore.
>
> **Metafora cliente-servitore**: dato un problema si fa uso di un insieme di centri di assistenza, ciascun centro è capace di eseguire un preciso insieme di attività. I clienti non conoscono l'organizzazione interna dei centri di servizio ne vi possono accedere direttamente.
>
> **Concetto di oggetto**: un centro di servizi con una parte nascosta e una parte visibile detta interfaccia costituita dalle operazioni che possono manipolare l'oggetto o che possiamo chiedere all'oggetto di eseguire.

Il C++ viene definito un linguaggio orientato agli oggetti e fornisce degli strumenti che consentono di creare unità funzionali il cui accesso viene controllato rigorosamente.

Vantaggi:

1. Il C++ esegue un controllo di correttezza migliore perché permette di organizzare meglio il codice e riutilizzare il codice esistente.
2. La programmazione ad oggetti rappresenta un nuovo modo di pensare che rende più facile definire un modello del problema.

Stream

Uno stream è una astrazione che si riferisce ad ogni flusso di dati da una sorgente (produttore) ad un ricevitore (consumatore).

#include <iostream.h> in C++ equivale a <stdio.h> del C

```
char *s = "Parola";
int i = 12;
double f = 34.56;
// << è l'operatore di inserimento e cout è il nuovo nome per stdout
cout << s << "  "<< i << " "<< d ;
```

Analogamente per l'input:

```
char s[80];
int i;
double d;
// >> è l'operatore di estrazione e cin è il nuovo nome di stdin
cin >> s >> i >> d;
```

Legge da console una stringa, un int e un double

Il tipo degli argomenti di >> determina il tipo di input che l'operatore aspetta.

L'operatore >> interrompe l'input quando trova uno spazio e non inserisce tale carattere nel buffer

Argomenti di default

In C++ i parametri a funzioni possono assumere valori di default che vengono assegnati nel caso in cui manchi il valore effettivo.
Esempio:

```
void print(int value, int base = 10);
void print(int value, int base) {          // base è un argomento di default
        cout << value << " " << base << endl;
}
print(45);              // Equivale a print(45, 10);
print(34, 56)           // Equivale a print(34, 56);
```

Gli argomenti di default devono sempre comparire alla estrema destra della lista dei parametri.
All'atto della chiamata i parametri di default possono mancare in numero qualsiasi, ma a partire dall'ultimo, cioè se una funzione ha tre parametri di default può essere chiamata passando tutti e tre i parametri, solo i primi due, solo il primo o nessuno ma non può essere chiamata, ad esempio, passando solo il terzo parametro.
Esempio:

```
void print(int value, int base = 10, int altezza);  // ERRORE
```

I valori che gli argomenti di default devono assumere vanno indicati nel prototipo della funzione e non nella definizione.
Il nome dei parametri di funzioni non usato localmente può essere omesso
Esempio:

```
void funz(int n, void*) {
// La funzione riceve un "void" che non essendo usato non ha un nome
        // Le istruzioni utilizzano solo "n"
}
funz(10, NULL);                 // La chiamata di funz() deve specificare tutti i parametri
```

Operatore di scope resolution

L'operatore di scope resolution **::** permette di indicare in quale contesto trovare una variabile
Esempio:

```
char ch = 'K';
void fhide() {
        char ch = 'W ';
        cout << "ch locale = " << ch << endl;
        cout << "ch globale = "<< ::ch << endl
        // L'operatore :: è usato per indicare la variabile ch più esterna possibile
}
```

L'operatore di scope resolution risolve i conflitti di visibilità
Esempio:

```
enum Forma { PICCOLA, MEDIA, NORMALE};
struct Contenitore {
        enum Forma { NORMALE, SUPER};.....};
struct Recipiente {
        enum Forma { SPECIALE, NORMALE };..... };
```

NORMALE può valere 2, 0, 1 a seconda dell'enum
É necessario qualificare l'enum da usare

```
Forma tipo_cart = NORMALE;
Contenitore::Forma tipo_cont = Contenitore::NORMALE;
Recipiente::Forma tipo_rec = Recipiente::NORMALE;
```

Ritorna

```
/* Default.cpp
 * Argomenti di default
 */
#pragma hdrstop
#include <condefs.h>
#include <iostream.h>

//---------------------------------------------------------------------------
#pragma argsused
int get_volume(int lunghezza, int larghezza = 2, int altezza = 3);
void Attesa(char *);
int main(int argc, char* argv[]) {
  int x = 10, y = 12, z = 15;

  cout << "Dati i lati ";
  cout << "il volume =  " << get_volume(x, y, z) << "\n";
  cout << "Dati i lati ";
  cout << "il volume =  " << get_volume(x, y) << "\n";
  cout << "Dati i lati ";
  cout << "il volume =  " << get_volume(x) << "\n";

  cout << "I lati della scatola sono:  ";
  cout << get_volume(x, 7) << "\n";
  cout << "I lati della scatola sono:  ";
  cout << get_volume(5, 5, 5) << "\n";
  Attesa("terminare");
  return 0;
}
int get_volume(int lunghezza, int larghezza, int altezza) {
  cout << "  " <<  lunghezza
       << "  " <<  larghezza
       << "  " <<  altezza
       << "  ";
  return lunghezza * larghezza * altezza;
}
void Attesa(char * str) {
  cout << "\n\n\tPremere return per " << str;
  cin.get();
}
```

Ritorna

```
/* Visib.cpp
 * Visibilità delle variabili
 */
#pragma hdrstop
#include <condefs.h>
#include <iostream.h>

//---------------------------------------------------------------------------
#pragma argsused
int x = 22;                        // x globale (x1)
void funz(int);
void Attesa(char *);
int main(int argc, char* argv[]) {
  cout << "Punto 1 x = " << x << " &x = " << &x <<endl;
  funz(x);
  cout << "Punto 9 x = " << x << " &x = " << &x << endl;
  Attesa("terminare");
  return 0;
}
void funz(int y) { // Il parametro y è locale a funz
  cout << "Punto 2 y = " << y << " &y = " << &y << endl;
  //int y;   // Errore y è definita due volte nello stesso spazio di visibilità
  int z = x;       // A z viene assegnato il valore della x globale (x1)
  cout << "Punto 3 z = " << z << " &z = " << &z << endl;
  int x;           // x locale (x2) nasconde x globale
  x = 1;           // 1 è assegnato all' x locale (x2)
  cout << "Punto 4 x = " << x << " &x = " << &x << endl;
  {
     int x;       // x (x3) nasconde il primo x locale
     x = 2;       // 2 è assegnato al secondo x locale (x3)
     cout << "Punto 5 x = " << x << " &x = " << &x << endl;
     ::x = 7;     // Tramite l'operatore scope resolution (::)
                  // 7 viene assegnato alla x globale (x1)
     cout << "Punto 6 x = " << ::x << " &x = " << &(::x)<< endl;
  }
  x = 3;          // 3 è assegnato al primo x locale (x2)
  cout << "Punto 7 x = " << x << " &x = " << &x << endl;
  ::x = 4;        // Tramite l'operatore scope resolution (::)
                  // 4 viene assegnato alla x globale (x1)
  cout << "Punto 8 x = " << ::x << " &x = " << &(::x)<< endl;
}
void Attesa(char * str) {
  cout << "\n\n\tPremere return per " << str;
  cin.get();
}
```

Riferimenti indipendenti

Un riferimento indipendente è un nome alternativo per un oggetto.
La notazione &X significa riferimento a X.
Esempio:

```
int A;
int &ar = A;
```

ar conterrà l'indirizzo di A; A questo punto esistono due modi diversi per riferirsi alla stessa variabile.
Quando si scrive:

```
int X = 10, Y = 100;
int &xr = X;
xr = Y;
```

non si assegna a xr l'indirizzo di Y, poiché il compilatore dereferenzia sempre l'indirizzo, ma copia il valore di Y in X.
I riferimenti non sono puntatori!
Un riferimento deve sempre essere inizializzato al momento della creazione e non può mai puntare a una diversa variabile..
Esempio:

```
int x = 10
int &r;
r = x;    // ERRORE: l'associazione a un riferimento deve avvenire in tempo di definizione
          // Viene automaticamente dereferenziata
int y = 10;
int *p;
p = &y;
          // OK: il valore di un puntatore può cambiare dopo la definizione
          // Il puntatore va esplicitamente dereferenziato
```

Soltanto in due casi è possibile avere un riferimento non inizializzato

1. Quando si dichiara una variabile esterna: extern int &X;
2. Quando esistono riferimenti membri di una classe, essi devono essere inizializzati nella lista di inizializzazione del costruttore, non nella definizione della classe.

Nota: L'uso dei riferimenti indipendenti sintatticamente corretto non rappresenta una buona tecnica di programmazione.

Parametri di funzione

Normalmente quando si passa un argomento a una funzione, la variabile che si specifica nella lista degli argomenti è copiata e usata dalla funzione.
Se si effettuano delle modifiche sul parametro, l'originale non viene cambiato.
Se si vuole cambiare la variabile originale, occorre indicare alla funzione dove la variabile è memorizzata, a tal scopo si può utilizzare un puntatore.
Un riferimento specificato dall'operatore & è il secondo metodo per passare un indirizzo ad una funzione.
Un riferimento assomiglia contemporaneamente a un puntatore (contiene un indirizzo) e a una variabile (non deve essere dereferenziato).
Se una funzione ha come argomento un riferimento:

```
void funcr(int &x);
```

essa viene chiamata nello stesso modo se si passasse un argomento per valore

```
fincr(a);
```

Il fatto di passare un indirizzo è trasparente.

I riferimenti di tipo void (void &) non sono permessi.
L'unico luogo in cui si notano i riferimenti, è nella lista di argomenti di una funzione; in qualunque altro luogo, nel corpo della funzione, l'argomento ha l'aspetto di una normale variabile.
Esempio:
Soluzione con i puntatori:

```
void swapp(int *x, int *y) {
        int tmp = *x;           // Devo dereferenziare
        *x = * y;
        *y = tmp;
}
int a = 10, b = 20;
swapp(&a, &b);
```

Soluzione con variabili "reference"

```
void swapr(int &x, int &y) {        // Vengono associate quando si passano i valori

        int tmp = x;
        x = y;
        y = tmp;
}
int a = 10, b = 20;
swapr(a, b);
```

Quando viene passato come argomento a una funzione un puntatore ad una struttura si selezionano i membri usando l'operatore '->'
Quando viene passato come argomento a una funzione un riferimento ad una struttura si selezionano i membri usando l'operatore '.'

Riferimenti costanti

Si possono anche creare riferimenti a valori costanti

```
int &aa = 47;
```

Il compilatore crea una variabile temporanea, copia in essa il valore costante, e usa l'indirizzo della variabile temporanea nel riferimento; ciò significa che qualunque funzione che ha un riferimento come argomento può anche avere un argomento costante.
La parola chiave "const" può essere usata per proteggersi da modifiche non intenzionali di variabili riferimento.
Esempio:

```
struct Cliente{....};
void CStampa(const Cliente &cli);      // "cli" non deve essere modificato da CStampa()
void CUpadate(Cliente &cli);   // "cli" può essere modificato
```

E' possibile passare a una funzione un argomento per riferimento e ritornare un riferimento a una variabile.
I riferimenti possono essere restituiti come "const" e come "non const"

Esempio:

```
static int count = 0;
int &GetCountR()   {
        return (count);
}
GetCountR() += 1;              // OK: GetCountR() può essere usata come l-value

const int &GetCountC()   {
        return (count);
}
GetCountC() += 1;              // ERRORE: GetCountC() non può essere l-value
```

Quando una funzione ritorna un indirizzo essa rende disponibile tale indirizzo all'utente, che può leggerne il valore e, se il puntatore non è di tipo const, anche di modificarne il valore.
In questo modo si lascia all'utente la possibilità di manipolare anche dati di tipo privato, compiendo una scelta progettuale che contrasta con la filosofia dei linguaggi a oggetti.

Vantaggi dei riferimenti:

Permettono di migliorare la chiarezza del codice, non ci si deve preoccupare dei dettagli relativi al passaggio dei parametri.
La responsabilità del passaggio degli argomenti a una funzione è demandata a colui che scrive le funzioni e non a colui che le utilizza
Sono un necessario complemento per l'overlodaing degli operatori
L'efficienza ottenibile passando degli indirizzi varia a seconda dei calcolatori.

Problemi con i riferimenti

Nei tipi di dati predefiniti gli elementi sono pubblici, quindi per rendere esplicito il tipo di passaggio degli argomenti, in modo che l'utente sappia che una variabile può essere modificata, si può usare un puntatore
Nei tipi di dati definiti dall'utente è possibile fare in modo che tutte le modifiche dei dati interni siano effettuate da funzioni membro, è più facile scoprire effetti collaterali nascosti.
I tipi definiti dall'utente sono in genere di dimensioni tali per cui il passaggio di un indirizzo è più efficiente del passaggio per valore.

Indicazioni per il passaggio degli argomenti

Se una funzione non modifica un argomento di un tipo predefinito o un tipo definito dall'utente piccolo, si passa l'argomento per valore
Se una funzione modifica un argomento di tipo predefinito si passa il puntatore
Se una funzione modifica un argomento di tipo definito dall'utente si passa il riferimento
Per passare oggetti non piccoli che non vengono modificati si passano riferimenti a costanti
Nel caso di parametri di grosse dimensioni, il passaggio per valore che implica una copia, diventa più lento e occupa una maggiore quantità di memoria.
Nel caso di parametri di dimensioni ridotte può risultare più piccolo del suo puntatore, il passaggio per valore è più rapido.
Il passaggio per riferimento permette di modificare la variabile passata come parametro ma crea un effetto collaterale e non bisogna abusarne.
Il passaggio per valore offre la possibilità di modificare e utilizzare il parametro senza modificare il valore originario.
Dato che il nome di un array è in effetti un puntatore, quando viene passato come parametro a una funzione è sempre passato per riferimento.

Ritorna

```
/* Alias.cpp
 * esempio di sinonimi, variabili reference
 */
#pragma hdrstop
#include <condefs.h>
#include <iostream.h>

//---------------------------------------------------------------------------
#pragma argsused
void Attesa(char *);
int main(int argc, char* argv[]) {
  int a = 2;
  int b = 10;

  int &c = b;   // c è un sinonimo per b

  cout << endl << "b vale " << b << endl;

  // l'istruzione che segue modifica il valore di b
  c = a * b;

  // b non dovrebbe essere cambiato (non è mai stato un lvalue ...)
  cout << endl << "adesso b vale " << b << endl;
  Attesa("terminare");
  return 0;
}
void Attesa(char * str) {
  cout << "\n\n\tPremere return per " << str;
  cin.get();
}
```

Ritorna

```
/* Passing.cpp
 * Passaggio di argomenti e ritorno di valori per valore
 * tramite puntatore e per riferimento
 */
#pragma hdrstop
#include <condefs.h>
#include <iostream.h>

//---------------------------------------------------------------------------
#pragma argsused
struct demo {
   int i, j, k;
};

demo byvalue(demo);          // Passaggio e ritorno per valore
demo *bypointer(demo *);     // Passaggio e ritorno con puntatore
demo &byreference(demo &);   // Passaggio e ritorno per riferimento
void print(demo, char *msg = "");
void Attesa(char *);
int main(int argc, char* argv[]) {
  demo B;
  B.i = B.j = B.k = 111;

  print(B,               "Valore nel main             ");
  print(byvalue(B),      "Risultato di byvalue(B)     ");
  print(B,               "Valore nel main             ");
  print(*bypointer(&B), "Risultato di bypointer(&B)  ");
  print(B,               "Valore nel main             ");
  print(byreference(B), "Risultato di byreference(B) ");
  print(B,               "Valore nel main             ");
  Attesa("terminare");
  return 0;
}
void print(demo d, char *msg) {
   cout << msg << ": i = " << d.i << " j = " << d.j
        << " k = " << d.k << endl << endl;
}

demo byvalue(demo X) {
   X.i = X.j = X.k = 999;
   return X;                  // Ritorna un oggetto
}

// In entrambe le funzioni bypointer() e byreference(), l'indirizzo dello
// argomento modificato viene ritornato poiche', qui e' l'unico oggetto "sicuro"
// poiche' e' un parametro formale).
// Non si dovrebbe mai ritornare l'indirizzo di una variabile locale.
demo *bypointer(demo *X) {
   X->i = X->j = X->k = 888;
   return X;                  // X in realta', in questa funzione e' un puntatore
}

// L'uso di X quando e' passato per riferimento e' identico a quando e' passato
// per valore
demo &byreference(demo &X) {
   X.i = X.j = X.k = 777;
   return X;                  // Ritorna un riferimento
}
void Attesa(char * str) {
  cout << "\n\n\tPremere return per " << str;
  cin.get();
}
/*
 * In ciascun caso la funzione print() e' chiamata per il valore di ritorno
 * della funzione, che e' un oggetto o l'indirizzo di un oggetto.
 * La sintassi di chiamata e' identica sia che si utilizzino i valori, sia che
 * si utilizzino i riferimenti.
 * L'unico luogo in cui si notano i riferimenti e' nella lista di argomenti di
 * una funzione, in qualunque altro luogo, nel corpo della funzione, l'argomento
 * ha l'aspetto di una normale variabile.
 */
```

Function Overloading

In generale è buona norma assegnare a funzioni diverse nomi diversi, ma quando alcune funzioni eseguono lo stesso compito su oggetti di tipi diversi, può essere conveniente utilizzare lo stesso nome.
Una funzione è identificata, "firma" dal nome e dall'elenco dei parametri.
La firma di una funzione è definita dalla sequenza dei parametri ricevuti e dal loro tipo.
Esempio:

```
int fputs(const char *s, FILE *fp);
        // La firma è (const char *s, FILE *fp)
double fabs (double x);
        // La firma è (double x)
void prova();
        // La firma è ()
```

Il tipo ritornato non fa parte della firma
Esempio:

```
double f1(int);
long f2(int);
        // Le due funzioni hanno la stessa firma
```

In C++ possono esistere funzioni con lo stesso nome purché abbiano firma diversa.
Esempio:

```
int max(int a, int b) {
        return (a > b ? a : b);
}
double max(double a, double b) {
        return (a > b ? a : b);
}
int x = max(10, 25);                    // Esegue max(int, int)
double y = max(12.4, 92,5);             // Esegue max(double, double)
double z = max(12, 34.6);               // ERRORE: situazione ambigua
        // Non converte perché la conversione è ambigua;
        // si deve convertire int in double oppure il double in int?
```

Ogni chiamata a una funzione overloaded richiede che ci sia una e solo una funzione effettiva con parametri corrispondenti da chiamare
Esempio:

```
f(int x, char c);
f(char c, int x);
```

La chiamata f(12, 15) è ambigua perché entrambe possono essere chiamate dopo una conversione standard, ambedue richiedono una conversione dello stesso tipo.

```
g(int x, char b, char c);
g(char x, int y, int x);
```

La chiamata g(13, 13, 115); è corretta perché la seconda funzione richiede una conversione mentre la prima ne richiede due, il compilatore determina che la seconda è la funzione più vicina.

Nota: Il compilatore controlla che le funzioni overloaded non siano ambigue solo quando viene incontrata una chiamata a tali funzioni e non in corrispondenza della loro definizione.

Gestione dei nomi

Il compilatore utilizza la tecnica di gestione dei nomi per codificare i nomi di funzioni e la firma in un unico nome per poter distinguere le funzioni overloaded.
Esempio:

int put(const char *s, FILE *fp);
Il compilatore genera un nome esterno visto dal link simile a:
put_FPCcP4FILE

La gestione dei nomi viene attivato anche se si definisce una sola funzione, questo può creare problemi di incompatibilità con altri linguaggi (linkando file oggetto ottenuti da compilatori diversi).
La direttiva extern "C" evita il la gestione die nomi, è utile quando una funzione deve essere richiamata da codice non C++
Esempio:

```
extern "C"put(const char *s, FILE *fp);
oppure:
extern "C" {
        #include <stdio.h>
        #include <stdlib.h>             // Non cambia il nome alle funzioni
}
```

Acquisire l’indirizzo di una funzione overloaded

Acquisire l’indirizzo di una funzione overloaded può introdurre ambiguità, poiché il compilatore può scegliere tra più funzioni; si supera l’ambiguità specificando i tipi di argomento del puntatore a funzione che conterrà l’indirizzo della funzione overloaded.
Esempio:

```
void funz(int);
void funz(float);
void (*fptr_int)(int) = funz;
void (*fptr_float)(float) = funz;
```

Nota: Due funzioni overloaded con gli stessi operatori ma con diversi tipi del risultato sono considerate come un errore di ridichiarazione.
Alcuni tipi sono considerati equivalenti perché lo stesso insieme di valori può essere usato per inizializarli
Esempio:

int f(const T);
int f(T);
La chiamata di f:
T f;
f(t); è considerata ambigua perché un oggetto di tipo T può essere usato al posto di un oggetto di tipo const T.

Ritorna

```
/* Overl2.cpp
 * overloading  di funzioni
 */
#pragma hdrstop
#include <condefs.h>
#include <iostream.h>

//---------------------------------------------------------------------------
#pragma argsused
int   demo(const int);     // Esegue il quadrato di un intero
int   demo(float);         // Esegue il triplo di un & restituisce un int
float demo(const float, float); // Esegue la media di due floats
void Attesa(char *);
int main(int argc, char* argv[]) {
  int   index  = 12;
  float length = 14.33;
  float height = 34.33;

  cout << "12 al quadrato =    " << demo(index)         << "\n";
  cout << "24 al quadrato =    " << demo(2 * index)     << "\n";
  cout << "Tre volte lengths = " << demo(length)        << "\n";
  cout << "Tre volte heights = " << demo(height)        << "\n";
  cout << "La media vale       " << demo(length,height) << "\n";
  Attesa("terminare");
  return 0;
}
int demo(const int in_value) {     // Esegue il quadrato di un intero
  return in_value * in_value;
}

int demo(float in_value) {   // Esegue il triplo di un float e restituisce un int
  return (int)(3.0 * in_value);
}

float demo(const float in1, float in2) { // Esegue la media di due floats
  return (in1 + in2)/2.0;
}
void Attesa(char * str) {
  cout << "\n\n\tPremere return per " << str;
  cin.get();
}
```

Definizione della classe

Una classe rappresenta un modo per combinare, in un'unica unità, blocchi di dati e funzioni per manipolarli.
Una classe è un tipo di dato e rappresenta un concetto statico, si trova scritto nel testo del programma.
Un oggetto è un'istanza del tipo di dato e rappresenta un concetto dinamico, si trova solo nel momento della esecuzione del programma.
La classe rende possibile definire un nuovo tipo di dato (tipo di dato astratto) attraverso la dichiarazione dei dati necessari per rappresentarli e delle operazioni che su di essi sono definite.
Tale tipo di dato viene trattato dal compilatore come uno dei tipi predefiniti, perché questo sia possibile occorre indicare al compilatore, definendo la classe, come trattare i dati.
La definizione di una classe è composta dal nome della classe seguito dal corpo, racchiuso tra parentesi graffe, e dal punto e virgola. Il corpo della classe contiene definizioni di variabili e dichiarazioni di funzioni, che possono essere usate soltanto in associazione con un oggetto che appartiene a quella classe.
Le variabili e le funzioni di una classe (membri) non sono accessibili da parte dell'utente e in tal caso vengono dette private.
Per permettere al programmatore che usa la classe, l'accesso ai membri della classe, si usa la parola public.
Esempio:

```
class prova {
        int i;          // Dato privato per default
public:
        void set(int v) { i = v; }
        int read() { return i; }
};
```

L’etichetta public separa il corpo della classe in due parti.
I nomi contenuti nella prima parte, privata, possono essere usati solo da funzioni membro.
La seconda parte, pubblica, costituisce l’interfaccia agli oggetti della classe.
Le funzioni membro set() e read() sono talvolta dette funzioni di accesso, poiché il loro unico scopo è quello di permettere l'accesso ai dati privati. Le funzioni membro possono essere chiamate solo se associate ad un oggetto;
Esempio:

```
prova A;
A.set(2);
int q = A.read();
```

Si può pensare ad un oggetto come ad una entità con uno stato interno e delle operazioni esterne (le funzioni membro).
Le chiamate di funzioni rappresentano i messaggi di un linguaggio orientato agli oggetti.
Il concetto di stato significa che un oggetto, quando non viene usato, ricorda delle informazioni che lo riguardano.
Uno dei vantaggi di progettare in C++, consiste nella separazione dell'interfaccia dalla implementazione.
L'interfaccia corrisponde alla definizione della classe, e descrive la forma degli oggetti e delle operazioni.
L'interfaccia di una classe è la sua dichiarazione:

```
nome
definizione dei dati interni
dichiarazione delle funzioni membro
```

Il corpo della classe è la collezione delle funzioni membro.
L'implementazione, che è composta da tutte le definizioni delle funzioni membro, mostra come vengono svolte le operazioni.
Se in futuro, il programmatore vuole migliorare l'implementazione, lo può fare senza dover modificare l'interfaccia e il codice compilato usando l'interfaccia.

L'incapsulamento dei dati permette una maggiore sicurezza vietando un accesso diretto ai dati privati della classe.
E' possibile progettare e codificare l'interfaccia, e ritardare la scrittura del codice della implementazione, usando l'interfaccia come se tale codice esistesse(solo il linker sa che non esiste).
Quando si crea una classe generalmente si vuole che esso sia facilmente accessibile e si vuole separare l'interfaccia dall'implementazione, in modo che l'implementazione possa essere modificata senza dover ricompilare l'intero sistema.
Si raggiunge questo scopo ponendo la dichiarazione della classe in un file header.
Anche le definizioni delle funzioni membro, è meglio che siano separate in un loro file; in questo modo possono essere compilate e verificate ed essere poi rese disponibili a chiunque voglia usare la classe come modulo o come libreria.
L'utente della classe deve semplicemente includere il file header, creare gli oggetti della classe e linkare il modulo oggetto o la libreria contenenti le definizioni delle funzioni della classe.

Puntatore this

Le funzioni membro (non definite come static) ricevono sempre un parametro nascosto; il puntatore this che punta all'oggetto sul quale lavorare.
La parola chiave this indica l'indirizzo iniziale dell'oggetto per cui la funzione è chiamata
this è implicitamente definito:

nome_classe *const this: e non può essere modificato.

This è un puntatore all'inizio della classe perciò si possono selezionare gli elementi con ->, ciò non è necessario all'interno di una funzione membro poiché è sufficiente riferirsi agli elementi con il loro nome.
return *this; produce una copia della classe come valore di ritorno.

Definizione di funzioni membro di una classe

I membri di una classe possono essere di tre categorie:

private	i dati non possono essere modificati dall'esterno e le funzioni non possono essere chiamate dall'esterno.
public	i dati sono accessibili liberamente dall'esterno e cosi pure le funzioni
protected	dati e funzioni sono visibili solo dalle classi ereditate.

Per definire una funzione membro occorre indicare al compilatore a quale classe è associata con l'operatore ::
L'operatore :: può essere usato anche per selezionare una definizione diversa da quella di default
L'operatore :: usato senza essere preceduto da un nome di classe, implica l'impiego del nome globale
Una funzione membro definita (non solo dichiarata) nella dichiarazione della classe viene considerata inline.
Una funzione membro può inoltre essere dichiarata inline all'esterno della dichiarazione della classe.

Esempio:

```
class stack {
        int size;
        char *top;
public:
        char pop();
};
inline char stack::pop() {
        return *--top;
}
```

Non è consentito fornire implementazioni diverse di funzioni membro inline in diversi file sorgenti poiché questo comprometterebbe la nozione di classe come tipo (singolo).

Regole per definire i dati membro:

- Elencati i servizi che una classe deve fornire all'esterno tutto il resto deve risultare invisibile.
- I dati non devono mai essere modificati dall'esterno
- I costruttori e i distruttori devono essere public
- Le funzioni per manipolazioni interne dei dati o invocate da altre funzioni della classe devono essere private
- Le altre funzioni saranno public

É possibile chiamare delle funzioni membro dall'interno di altre funzioni membro.

Se si sta definendo una funzione membro, tale funzione è già associata ad un oggetto (l'oggetto corrente, a cui ci si riferisce con la parola chiave this), una funzione membro può dunque essere chiamata usando soltanto il suo nome

Membri di tipo static di una classe

Ogni volta che si definisce un oggetto appartenente ad una particolare classe, tutti i dati della classe vengono duplicati.

É possibile definire una variabile in una classe, in modo che venga creata una sola istanza della variabile per tutti gli oggetti definiti in quella classe; tutti gli oggetti hanno accesso ai dati, che sono condivisi tra di essi; per fare questo basta dichiarare la variabile come static.

Si utilizzano le variabili di tipo static per comunicare tra oggetti.

Gli oggetti locali dichiarati static vengono creati e inizializzati una volta sola ma sono utilizzabili sono nel blocco in cui sono dichiarati.

Occorre riservare esplicitamente la memoria e inializzare tutti gli oggetti di tipo static; la memoria non viene allocata automaticamente poiché serve un'unica allocazione di memoria per l'intero programma.

La definizione di una variabile static è composta dal tipo dell'oggetto, dal nome della classe seguito dall'operatore :: (indica l'area di visibilità della variabile), e dall'identificatore della variabile.

La definizione e inizializzazione deve trovarsi fuori da ogni classe e corpo di funzione e deve essere unica.

Oggetti statici sono costruiti prima di eseguire il main()

Oggetti statici vengono distrutti dopo che l'ultima istruzione del main() è stata eseguita

Esempio:

```
static Animale animale1;                    // Costruito prima di eseguire il main()
int main() {
        static Animale animale2;            // Costruito prima di eseguire il main()
        ........
        if(a == b) {
                static Animale animale3;    // Costruito prima di eseguire il main()
                .......
        }
}                       // Distrugge animale1, animale2, animale3
```

Membri di tipo const di una classe

É possibile rendere un membro di una classe di tipo const, il significato è che un oggetto, durante la sua esistenza, rimane costante; una costante all'interno di una classe occupa sempre memoria.
Una costante deve essere inizializzata all'interno di un costruttore, ciò avviene nella lista di inizializzazione del costruttore, dopo la lista degli argomenti, ma prima del corpo del costruttore.
Le funzioni membro const non modificano l'oggetto sul quale lavorano
Dire che una funzione membro è const equivale a dire che this punta ad un'area const; in tal caso this è definito come:

```
const nome_classe *const this;
```

La parola chiave const fa parte della firma di una funzione membro.
Oggetti const possono essere manipolati solo da funzioni membro const.
Esempio:

```
class Animale {
public:
        ......
        void Display() const;
        void Setnome(char *nome);
private:
        ......
};
void funz(const Animale &animale) {    // Reference a oggetto const
        animale.Display();                      // OK: Display() è una member function const
        animale.Setnome("Pluto");
        // Errore: Setnome() è una member function non-const
```

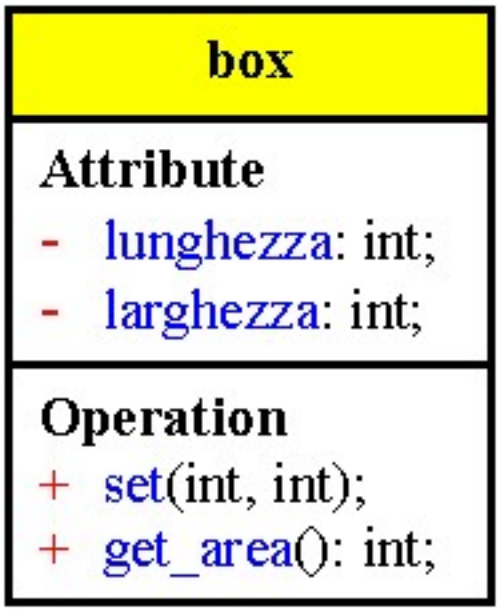
Ritorna

```
/* PBox.cpp
 * Utilizzo della classe box
 */
#pragma hdrstop
#include <condefs.h>
#include <iostream.h>
#include "box.h"
//---------------------------------------------------------------------------
USEUNIT("Box.cpp");
//---------------------------------------------------------------------------
#pragma argsused
void Attesa(char *);
int main(int argc, char* argv[]) {
  box piccolo, medio, grande;          //Instanzia tre box

  piccolo.set(5, 7);
  medio.set(10, 14);
  grande.set(15, 21);

  cout << "L'area del box piccolo e'  = " << piccolo.get_area() << "\n";
  cout << "L'area del box medio e'    = " << medio.get_area()   << "\n";
  cout << "L'area del box grande e'   = " << grande.get_area()  << "\n";
  Attesa("terminare");
  return 0;
}
void Attesa(char * str) {
  cout << "\n\n\tPremere return per " << str;
  cin.get();
}
```

```
/* Box.h
 */
//---------------------------------------------------------------------------
#ifndef BoxH
#define BoxH
class box {
   int lunghezza;
   int larghezza;
public:
   void set(int new_length, int new_width);
   int get_area(void) {return (lunghezza* larghezza);}
};
//---------------------------------------------------------------------------
#endif
```

```
//---------------------------------------------------------------------------
/* Box.cpp
 */
//---------------------------------------------------------------------------
#pragma hdrstop

#include "Box.h"
//---------------------------------------------------------------------------
#pragma package(smart_init)
// Questo metodo setta box ai parametri impostati
void box::set(int new_length, int new_width) {
  lunghezza = new_length;
  larghezza = new_width;
}
```

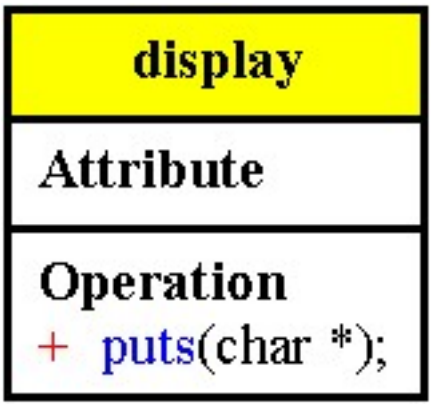
Diagramma U. M. L. delle classi



Ritorna

```
/* Display.cpp
 * Una classe con una propria funzione puts()
 */
#pragma hdrstop
#include <condefs.h>
#include <iostream.h>

//---------------------------------------------------------------------------
#pragma argsused
class display {
public:
  void puts(char *);            // Dichiara la funzione
};
void Attesa(char *);
int main(int argc, char* argv[]) {
  display A;
  A.puts("Chiama ::puts()");
  Attesa("terminare");
  return 0;
}
void display::puts(char * msg) {
  ::puts("All'interno della mia funzione puts");
  ::puts(msg);
}
void Attesa(char * str) {
  cout << "\n\n\tPremere return per " << str;
  cin.get();
}
/*
 * L'operatore :: puo' essere usato per selezionare una definizione diversa da
 * quella di default.
 * L'operatore :: usato senza essere preceduto da un nome di classe, implica
 * l'impiego del nome globale
 */
```

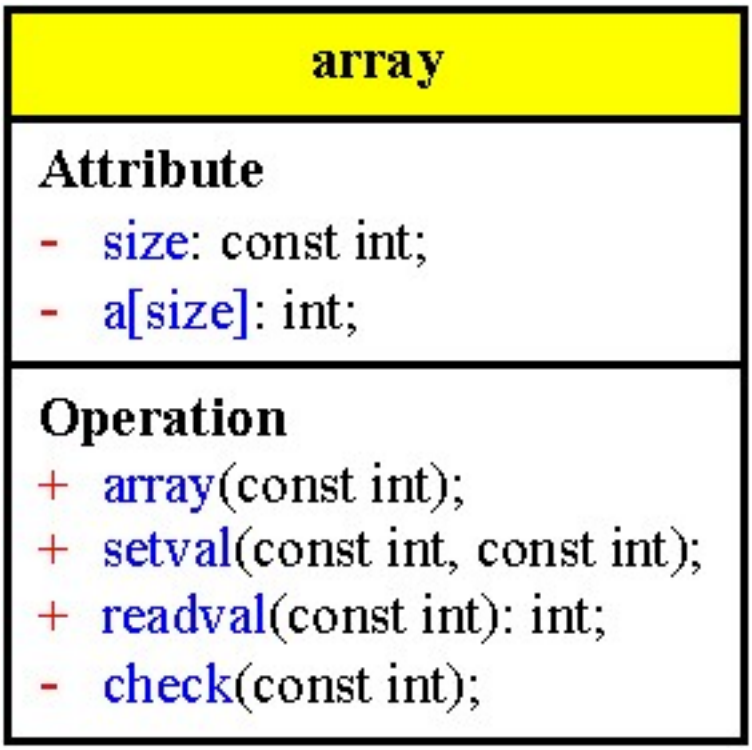
Diagramma U.M.L. delle classi



Ritorna

```
/* Smart.cpp
 * Un array che controlla gli estremi
 */
#pragma hdrstop
#include <condefs.h>
#include <iostream.h>

//---------------------------------------------------------------------------
#pragma argsused
class array {
  enum { size = 10};
  int a[size];
  void check(const int index);   // Funzione privata
public:
  array(const int initval = 0);        // Valore di default dell'argomento
  void setval(const int index, const int value);
  int readval(const int index);
};

void Attesa(char *);
int main(void) {
  array A, B(47);
  int x = B.readval(10);               // Fuori limite -- vedere cosa succede

  Attesa("terminare");
  return 0;
}
// Costruttore non duplica il valore di default
array::array(const int intval) {
  for (int i = 0; i < size; i++)
    setval(i, intval);                // Chiama un'altra funzione membro
}

void array::check(const int index) {
  if(index < 0 || index == size) {
    cerr << "Errore array: indice setval fuori limite" << endl;
    Attesa("terminare");
    exit(1);            // Funzione di libreria ANSI C; esce dal programma
  }
}

void array::setval(const int index, const int value) {
  check(index);
  a[index] = value;
}

int array::readval(const int index) {
  check(index);
  return a[index];
}

void Attesa(char * str) {
  cout << "\n\n\tPremere return per " << str;
  cin.get();
}
/*
 * E' possibile chiamare delle funzioni membro dall'interno di altre funzioni
 * membro.
 * Se si sta' definendo una funzione membro, tale funzione e' gia' associata ad
 * un oggetto l'oggetto corrente (this), puo' quindi essere chiamata usando
 * soltanto il suo nome.
 * check(),
 * e' una funzione membro di tipo private, che puo' essere chiamata
 * solo da un'altra funzione membro.
 */
```

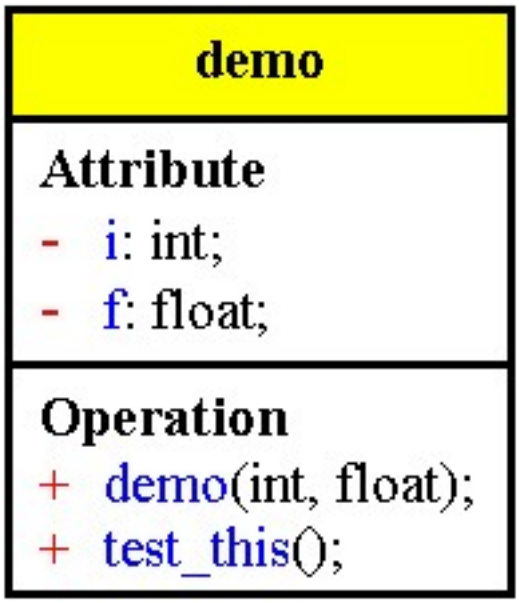
Diagramma U. M. L. delle classi



Ritorna

```
/* This.cpp
 * Illustra la parola chiave "this"
 */
#pragma hdrstop
#include <condefs.h>
#include <iostream.h>

//---------------------------------------------------------------------------
#pragma argsused
class demo {
  int i;
  float f;
public:
  demo(int ii = 0, float ff = 0.0) {
    i = ii;
    f = ff;
  }
  void test_this() {
    cout << "i = " << this->i << endl;
    cout << "f = " << this->f << endl;
  }
};

void Attesa(char *);
int main(int argc, char* argv[]) {
  demo X(1957, 7.865);
  X.test_this();
  Attesa("terminare");
  return 0;
}
void Attesa(char * str) {
  cout << "\n\n\tPremere return per " << str;
  cin.get();
}
/*
 * La parola chiave this indica l'indirizzo dell'oggetto per cui la funzione
 * e' stata chiamata.
 */
```

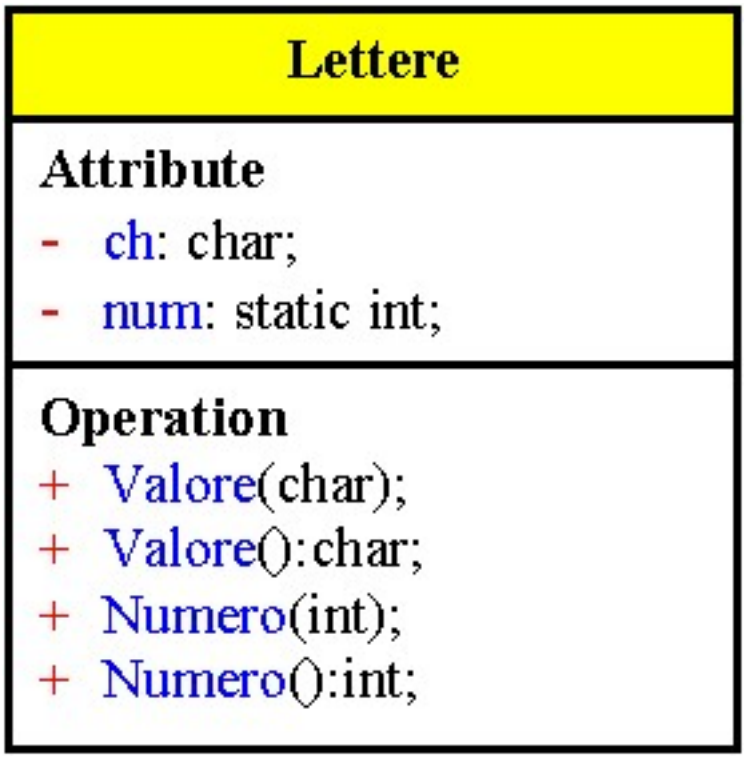
Diagramma U. M. L. delle classi



Ritorna

```
/* Letters.cpp
 * esempio di classe con dati di classe e dell'uso dei distruttori
 * Con membri static
 */
#pragma hdrstop
#include <condefs.h>
#include <iostream.h>

//---------------------------------------------------------------------------
#pragma argsused
// definizione della classe Lettere
class Lettere {
  // dati privati:
  char          ch;
  static int    num;

public:
  void  Valore(char c) { ch = c; }
  char  Valore ()      { return ch; }
  static void Numero(int n) { num = n; }
  static int  Numero ()     { return num; }
};
// definizione "esterna" della variabile di classe
int Lettere :: num = 0;
// esempio di utilizzo della classe
void Attesa(char *);
int main(int argc, char* argv[]) {
  // la seguente chiamata a una funzione statica si
  // riferisce alla classe nel suo insieme, e non
  // ad un oggetto specifico
  cout << "Numero di lettere all'inizio: "
       << Lettere::Numero() << endl;

  // creazione di due oggetti di tipo lettere
  Lettere A, B;
  A.Valore('A');
  B.Valore('B');
  cout << "Valore di A = " << A.Valore() << endl;
  cout << "Valore di B = " << B.Valore() << endl;
  A.Numero(5);
  cout << "Numero di A = " << A.Numero() << endl;
  cout << "Numero di B = " << B.Numero() << endl;

  Attesa("terminare");
  return 0;
}
void Attesa(char * str) {
  cout << "\n\n\tPremere return per " << str;
  cin.get();
}
```

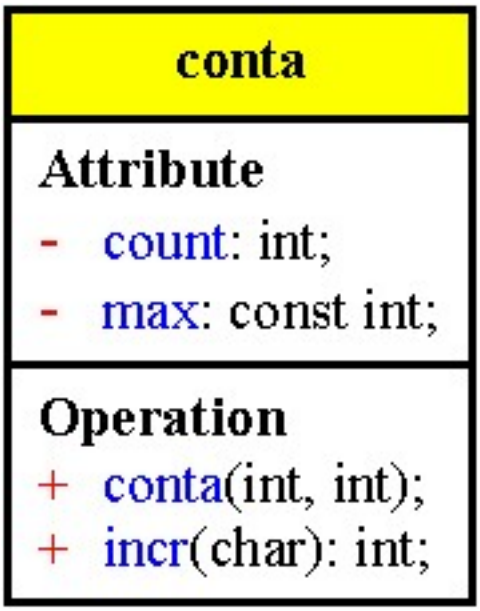
Diagramma U. M. L. delle classi



Ritorna

```
/* Constmem.cpp
 * Membri costanti di una classe
 */
#pragma hdrstop
#include <condefs.h>
#include <iostream.h>

//---------------------------------------------------------------------------
#pragma argsused
class conta {
  int count;
  const int max;           // int e' opzionale
public:
  conta(int Max = 7, int init = 0) : max(Max) {
    count = init;
  }
  int incr(char ch) {
    cout << ch << "\t" << count+1 << "\t" << ((count+1) == max) << endl;
    if (++count == max)
      return 1;
    return 0;
  }
};

void Attesa(char *);
int main(int argc, char* argv[]) {
  conta A, B(9), C(3,2);
  while( !B.incr('B'))
    if(A.incr('A'))
      C.incr('C');
  Attesa("terminare");
  return 0;
}
void Attesa(char * str) {
  cout << "\n\n\tPremere return per " << str;
  cin.get();
}
/*
 * Il significato di costante per i membri di una classe, e' che un oggetto
 * durante la sua esistenza rimane costante.
 * Una costante deve essere inizializzata dal costruttore
 */
```

Diagramma U.M.L. delle classi



Costruttori e distruttori

Il C++ permette di assicurare che le classi siano sempre inizializzate e distrutte correttamente, per fare ciò il compilatore ha bisogno di una funzione da chiamare quando l'oggetto viene creato (costruttore) e di una funzione da chiamare quando l'oggetto diventa non più visibile (distruttore).
Il costruttore è una funzione membro con lo stesso nome della classe, che alloca la memoria per tutte le variabili dell'oggetto e viene invocato quando si inizializzano gli oggetti.
Una classe può avere molti costruttori ma può anche non averne nessuno, in questo caso il compilatore aggiunge un costruttore di default senza parametri che inizializza tutti i membri a zero.
Se pero' si aggiunge un costruttore di qualsiasi tipo allora il compilatore non aggiunge più il costruttore di default.
Un costruttore non restituisce alcun valore.
Il nome del distruttore è uguale al nome della classe con il carattere tilde (~) iniziale; viene invocato automaticamente quando l'oggetto esce dal suo intervallo di visibilità.
I distruttori sono funzioni membro che hanno lo scopo di rilasciare le risorse acquisite.
Una classe può avere solo un distruttore.
Il distruttore non ha mai argomenti, viene chiamato solo dal compilatore, e non può essere chiamato esplicitamente dal programmatore.
Esempio:

```
class stack {
        int max;
        int sp;
        float *stk;
public:
        stack(int size);      // Costruttore
        ~stack();             // Distruttore
};
stack::stack(int size) {    // Definizione del costruttore
        stk = new float[max = size];
        sp = 0;
}
stack:: ~stack() {                  // definizione del distruttore
        delete [] stk;
}
```

Un oggetto viene creato quando si incontra la sua definizione e distrutto quando non è più visibile.

Costruttore di default

Il costruttore di default è un costruttore che può essere eseguito senza parametri.

```
class Animale{
public:
        Animale();
.....};
oppure:
class Animale {
public:
        Animale(const char *nome = 0);
.....};
```

Il costruttore di default viene eseguito quando si istanzia un oggetto senza inizializzarlo in modo esplicito.

```
static Animale a1;
Animale a2;
Animale *p = new Animale;
```

Una classe può avere un solo costruttore di default.
Se la classe non ha altri costruttori, il compilatore genera un costruttore di default che non fa niente.
Il costruttore di default è necessario quando i membri sono dei puntatori.
Nota: L'esecuzione dei distruttori è garantita solo se si esce da main() usando return; l'uscita con exit() potrebbe inibire la distruzione degli oggetti sullo stack.
Secondo l'ANSI C++ un main() senza return equivale a un main() con return(0)

Inizializzazione aggregata

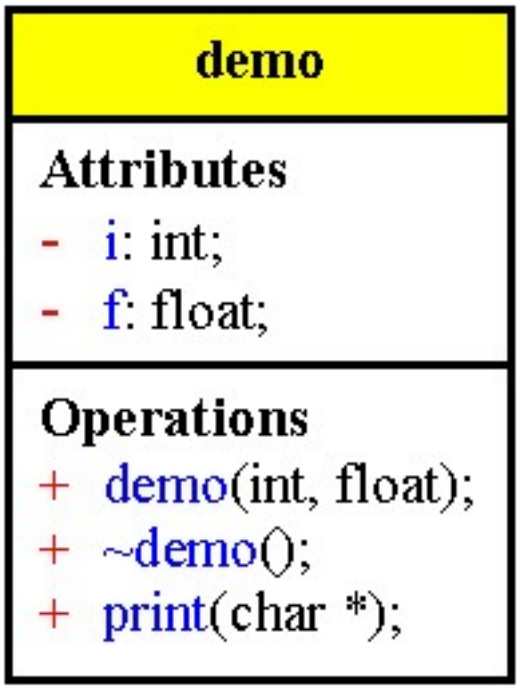
I valori di inizializzazione vengono copiati negli elementi della struttura nell'ordine di apparizione
la struttura senza il nome del tipo, non è necessario poiché viene definita immediatamente una istanza.
Anche le variabili static possono utilizzare l'inizializzazione aggregata.
L'inizializzazione aggregata di un array di oggetti con membri privati avviene chiamando il costruttore
La definizione dell'array non contiene la dimensione che viene calcolata automaticamente dal compilatore.
Definire un array di oggetti senza una lista di inizializzazione aggregata è possibile:

- solo se la classe non ha dati o funzioni privati
- se possiede un costruttore di default (senza argomento o con argomenti di default)

Ritorna

```
/* Costrut.cpp
 * Una classe con i costruttori
 */
#pragma hdrstop
#include <condefs.h>
#include <iostream.h>

//---------------------------------------------------------------------------
#pragma argsused
class demo {
  int i;
  float *f;
public:
  // Costruttore
  demo(int q, float p) {
    i = q;
    f = new float(p);
  }
  // Distruttore
  ~demo() { delete f; }
  void print(char *msg) {
    cout << msg     << ": " << endl;
    cout << "i  = " << i    << endl;
    cout << "*f = " << *f   << endl;
  }
};

void Attesa(char *);
int main(void) {
  demo A(9, 11.47);   // Chiamata del costruttore
  A.print("A -- Costruttore");
  Attesa("terminare");
  return 0;
} // Distruzione di A
void Attesa(char * str) {
  cout << "\n\n\tPremere return per " << str;
  cin.get();
}
/*
 * Il compilatore ha bisogno di una funzione da chiamare quando la variabile
 * viene creata (costruttore) e di una funzione da chiamare quando la variabile
 * diventa non piu' visibile (distruttore).
 * Il costruttore e' una funzione membro con lo stesso nome della classe, che
 * alloca la memoria per tutte le variabili dell'oggetto.
 * Il nome del distruttore e' uguale al nome della classe con il carattere ~
 * iniziale
 */
```

Diagramma U. M. L. delle classi



Ritorna

```
/* Maincont.cpp
 * main() del contatore
 */
#pragma hdrstop
#include <condefs.h>
#include <iostream.h>
#include "Contator.h"
//---------------------------------------------------------------------------
USEUNIT("Contator.cpp");
//---------------------------------------------------------------------------
#pragma argsused
void Test1();
void Test2();
void ShowCount(counter c0);
void Test3();
void ShowCountRef(counter &c0);
void Test4();
counter TheSame(counter c0);
void Test5();
void Test6();
void Test7(counter c0);
void Test8();
void Test9();
void Test10();
void Test11();
void Test12();
void Test13();
void Test14();
void Test15();
void Test16();
void Test17();
void Attesa(char *);
int main(int argc, char* argv[]) {
  int sc;
  do {
    {
      counter c00;
      c00.AzzeraNum();
    }
    clrscr();
    cout << "Numero del Test<1-17> -- (0 per terminare)" << endl;
    cin  >> sc;
    switch (sc) {
      case 0  : break;
      case 1  : Test1();  break;
      case 2  : Test2();  break;
      case 3  : Test3();  break;
      case 4  : Test4();  break;
      case 5  : Test5();  break;
      case 6  : Test6();  break;
      case 7  : {counter c01;
                 Test7(c01);
                 cout << "main numero contatori " << c01.NumObj() << endl;
                }
                break;
      case 8  : Test8();  break;
      case 9  : Test9();  break;
      case 10 : Test10(); break;
      case 11 : Test11(); break;
      case 12 : Test12(); break;
      case 13 : Test13(); break;
      case 14 : Test14(); break;
      case 15 : Test15(); break;
      case 16 : Test16(); break;
      case 17 : Test17(); break;
      default : cout << "\n\n\t\t\tTest non previsto\n";
    }
    Attesa("continuare");
  } while(sc);
  return 0;
}
void Test1() {
  cout << "\nUtilizzo delle funzioni membro Reset(), Inc()\n\n";
  counter c1, c2;
  finedefs;
  c1.Reset();
```

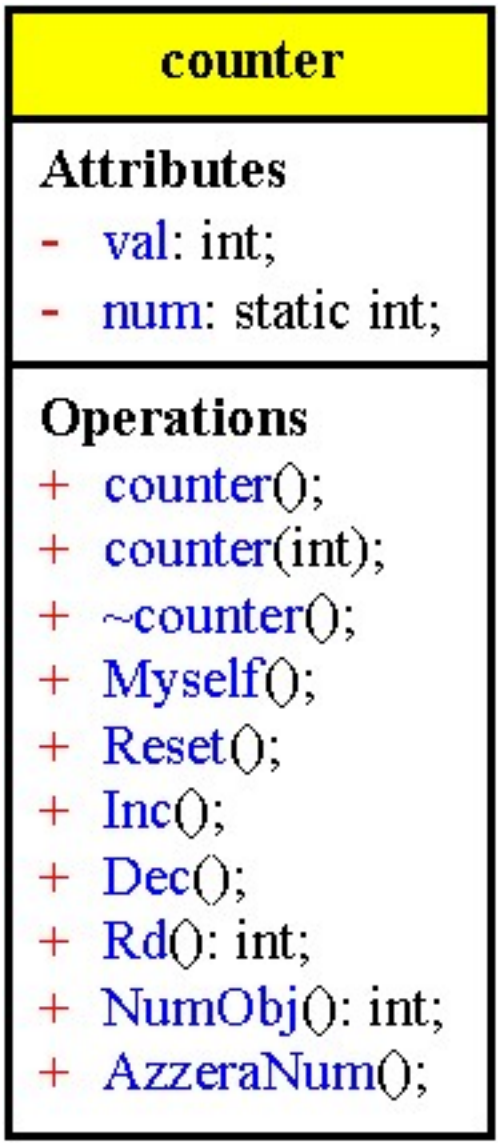
```
  c2.Reset();
  c1.Inc();
  cout << "Valore dei contatori\t" << c1.Rd() << "\t" << c2.Rd() << endl;
  fineblocco;
}
void Test2() {
  cout << "\nUtilizzo della funzione Myself()\n\n";
  counter c1;
  counter c2(12);
  finedefs;
  c1.Myself();
  c2.Myself();
  fineblocco;
}

void ShowCount(counter c0) {
  c0.Myself();
}
void Test3() {
  cout << "\nPassaggio per valore alla funzione ShowCount()\n\n";
  counter c1(10);
  counter c2(10);
  finedefs;
  c1.Reset();
  c2.Inc();
  ShowCount(c1);
  ShowCount(c2);
  fineblocco;
}
void ShowCountRef(counter &c0) {
  c0.Myself();
}
void Test4() {
  cout << "\nPassaggio per riferimento alla funzione ShowCountRef()\n\n";
  counter c1(10);
  counter c2(10);
  finedefs;
  c1.Reset();
  c2.Inc();
  ShowCountRef(c1);
  ShowCountRef(c2);
  fineblocco;
}
counter TheSame(counter c0) {
  return c0;
}
void Test5() {
  cout << "\nValore di ritorno della funzione TheSame()\n\n";
  counter c1(20);
  finedefs;
  cout << "Valore di TheSame(c1).Rd();\t" << TheSame(c1).Rd() << endl;
  fineblocco;
}
void Test6() {
  cout << "\nUtilizzo della variabile di classe num\n\n";
  counter c1;
  cout << "Numero contatori\t" << c1.NumObj() << endl;
  counter c2(12);
  cout << "Numero contatori\t" << c2.NumObj() << endl;
  counter c3(10);
  cout << "Numero contatori\t" << c3.NumObj() << endl;
  fineblocco;
}
void Test7(counter c0) {
  cout << "\nCostruzione per copia  al trasferimento per valore\n"
          " di un oggetto ad una funzione\n\n";
  counter c2(12);
  finedefs;
  cout << "Test7 numero contatori " << c2.NumObj() << endl;
  fineblocco;
}
void Test8() {
  cout << "\nCostruzione per copia nella definizione di un oggetto\n"
          "la cui configurazione iniziale si vuole sia identica a quella\n"
          "di un oggetto gia' costruito\n\n";
  counter c1(10), c2(c1);
```

```
  finedefs;
  c2.Myself();
  fineblocco;
}
void Test9() {
  cout << "\nAllocazione di memoria per l'oggetto puntatore\n"
          "e invocazione del costruttore\n\n";
  counter *pc0;
  finedefs;
  pc0 = new counter(7);
  pc0->Myself();
  delete pc0;
  fineblocco;
}
void Test10() {
  cout << "\nAllocazione di memoria per l'oggetto puntatore\n"
          "e utilizzo del costruttore per copia\n\n";
  counter c1(10), *pc0 = new counter(c1);
  finedefs;
  pc0->Myself();
  cout << "Test10: numero contatori " << pc0->NumObj() << endl;
  delete pc0;
  fineblocco;
}
void Test11() {
  cout << "\nUtilizzo dei puntatori\n\n";
  counter c1, *pc0;
  finedefs;
  pc0 = &c1;
  c1.Myself();
  pc0->Myself();
  fineblocco;
}
void Test12() {
  cout << "\nOggetti costanti\n\n";
  const counter c1;
  finedefs;
  c1.Inc();
  c1.Myself();
  cout << "Valore = " << c1.Rd() << endl;
  fineblocco;
}
void Test13() {
  cout << "\nOggetti costanti e utilizzo dei puntatori\n\n";
  const counter c1;
  const counter *pc1;
  finedefs;
  pc1 = &c1;
  pc1->Inc();
  pc1->Myself();
  cout << "Valore = " << pc1->Rd() << endl;
  fineblocco;
}
void Test14() {
  cout << "\nOggetti costanti e utilizzo dei puntatori\n\n";
  const counter c1;
  const counter *pc1; // deve essere un puntatore a un oggetto costante
  finedefs;
  pc1 = &c1;
  pc1->Myself();
  //delete pc1;      // Attenzione!
  fineblocco;
}
void Test15() {
  cout << "\nArray di oggetti\n\n";
  counter c1[] = {counter(10), counter(20) };
  finedefs;
  for(int i = 0; i < sizeof(c1)/sizeof(counter); i++)
    c1[i].Myself();
  fineblocco;
}
void Test16() {
  cout << "\nArray di oggetti\n\n";
  counter c1[2];
  finedefs;
  for(int i = 0; i < 2; i++)
```

```
    c1[i].Myself();
   fineblocco;
}
void Test17() {
  cout << "Array di oggetti\n\n";
  counter *c1 = new counter[2];
  finedefs;
  for(int i = 0; i < 2; i++)
    c1[i].Myself();
  delete[]c1;
  fineblocco;
}
void Attesa(char * str) {
  cout << "\n\n\tPremere return per " << str;
  cin.ignore(4, '\n');
  cin.get();
}
```

```
/* Contator.h
 * classe counter
 */
//---------------------------------------------------------------------------
#ifndef ContatorH
#define ContatorH
#include <conio.h>
#define finedefs   cout << "=============== Fine definizione" << endl
#define fineblocco cout << "=============== Fine blocco"      << endl

class counter {
  int val;
  static int num;
public:
  counter();
  counter(int v);
  ~counter();
  void Myself();
  void Reset();
  void Inc();
  void Dec();
  int  Rd()const;
  int NumObj()     { return num; }
  void AzzeraNum() { num = 1;    }
};
//---------------------------------------------------------------------------
#endif


//---------------------------------------------------------------------------
/* Contator.cpp
 * Classe contatore
 */
//---------------------------------------------------------------------------
#pragma hdrstop
#include <iostream.h>
#include "Contator.h"
//---------------------------------------------------------------------------
#pragma package(smart_init)
int counter::num = 0;
counter::counter() {
  cout << "Eseguo il costruttore_0 su\t" << this << endl;
  val = 0; num++;
}
counter::counter(int v) {
  cout << "Eseguo il costruttore_1 su\t" << this << endl;
  val = v; num++;
}
counter::~counter() {
  cout << "Eseguo il distruttore su\t" << this << endl;
  num--;
}
void counter::Myself() {
  cout << "Contatore di indirizzo\t" << this << "\t"
       << "con valore\t" << Rd()
       << endl << "Cui giungo anche con this->val\t" <<  this->val << endl;
}
void counter::Reset() { val = 0;    }
void counter::Inc()   { val++;      }
void counter::Dec()   { val--;      }
int  counter::Rd()const  { return val; }
```

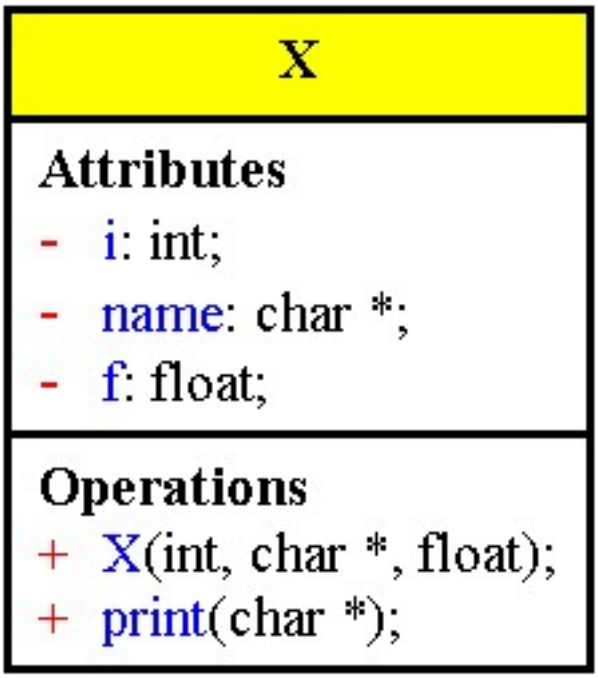
Diagramma U. M. L. delle classi



Ritorna

```
/* Aggreg.cpp
 * Inizializzazione di aggregati
 */
#pragma hdrstop
#include <condefs.h>
#include <iostream.h>

//---------------------------------------------------------------------------
#pragma argsused
class X  {
  int i;
  char *name;
  float f;
public:
  X(int ii = 0, char *nm = "", float ff = 0.0) {
    i = ii;
    name = nm;
    f = ff;
  }
  void print(char *id = "") {
    if(*id)
      cout << id << " ";
    cout << "i = " << i << " nome = " << name << " f = " << f << endl;
  }
};

void Attesa(char *);
int main(int argc, char* argv[]) {
  X z[] = { X(1, "1", 1.1), X(2, "2", 2.2), X(3, "3", 3.3) };
  const zsize = sizeof(z)/sizeof(z[0]);
  for(int i = 0; i < zsize; i++)
    z[i].print();
  cout << endl;
  // E' possibile dichiarare un array di oggetti senza una lista di
  // inizializzazione di aggregati solo se la classe ha un costruttore
  // senza argomenti o con argomenti di default
  X arr[10];
  arr[2].print();

  Attesa("terminare");
  return 0;
}
void Attesa(char * str) {
  cout << "\n\n\tPremere return per " << str;
  cin.get();
}
/*
 * L'inizializzazione aggregata per un array di oggetti con membri privati
 * ciascun elemento viene inizializzato chiamando il costruttore
 * Se si vuole cambiare il numero di elementi dell'array e' sufficiente
 * aggiungere o eliminare alcuni elementi dalla lista di inizializzazione
 */
```

Diagramma U. M. L. delle classi



Creazione dinamica di oggetti

Il C++ oltre a permettere la creazione di oggetti con durata di vita dipendente dalla visibilità (oggetti basati sullo stack), permette la creazione di oggetti con durata di vita arbitraria (oggetti basati sulla heap)
Quando si crea dinamicamente un oggetto in C++ si ottiene un nuovo oggetto inizializzato.
Quando si definisce un oggetto a tempo di compilazione il compilatore genera codice che decrementa il puntatore allo stack per creare sullo stack lo spazio sufficiente per contenerlo.
Tutta la memoria necesaria a un blocco viene allocata all'inizio del blocco anche se le variabili sono definite dopo.
La memoria viene sempre creata ogni volta che una funzione viene chiamata. quando un blocco termina il compilatore genera codice per spostare il puntatore allo stack indietro, e rimuovere così dallo stack tutte le variabili create per quel blocco.
Dalla parte opposta dello spazio di memoria allocato per il programma si trova la memoria libera chiamata heap.
In C++ il concetto di allocazione dinamica della memoria è stato esteso e incorporato nel nucleo del linguaggio.
Le parole chiave

new alloca la memoria e chiama il costruttore
delete distrugge l'oggetto e chiama il distruttore

New e delete gestiscono puntatori.
Esempio:

```
struct Cliente { .... };
Cliente *p;                 // Puntatore a Cliente
if((p = new Cliente) == 0)
        cout << "Manca memoria! << endl;
.....
delete p;
```

I nuovi operatori new e delete sono da preferire per allocare la memoria dinamica.
Le funzioni malloc(), calloc() e free() possono essere usate in C++ anche se sconsigliate
Esempio:
Allocazione usando malloc / free

```
Cliente *p;
p = (Cliente *) malloc(sizeof(Cliente)); // Non esegue il costruttore
.....
free(p);                                 // Non esegue il distruttore
```

Allocazione usando new / delete

```
Cliente *p = new Cliente;                // Esegue il costruttore
.....
delete p;                                // Esegue il distruttore
```

É possibile intercettare qualsiasi fallimento dell'operatore new usando set_new_handler()
Esempio:

```
void manca_memoria() {
        cout << "Manca memoria!"<< endl;
        exit(1);
}
main() {
        set_new_handler(manca_memoria);
                // Da qui in avanti, se l'operatore new fallisce,
                // viene richiamata la funzione manca_memoria()   }
```

Oggetti di dimensione arbitraria

Quando utilizzare l'allocazione dinamica:

- Non si conosce quanti oggetti saranno necessari a tempo di esecuzione.
- Non si conosce la durata di vita di un oggetto
- Non si può conoscere, fino a tempo di esecuzione, la quantità di memoria di cui un oggetto avrà bisogno.

Quando si usa new per creare un oggetto, viene allocata sullo heap una quantità di memoria sufficiente per i dati dell'oggetto e poi viene chiamato il costruttore per l'oggetto.

Array di oggetti

Se si alloca un array di oggetti , senza inizializzazione aggregata, la classe a cui gli oggetti appartengono deve contenere un costruttore senza argomenti, chiamato costruttore di default

Esempio:

```
object *op = new object[10];
```

L'istruzione delete []op; indica al compilatore di chiamare il distruttore per ciascun oggetto nell'array.

Quando si usa new, si deve passare come argomento il tipo di dato che si vuole creare in modo che l'operatore sappia quanta memoria allocare.

Quando si chiama delete si deve solo passare un indirizzo iniziale poiché new mantiene traccia della memoria allocata.

New memorizza la quantità di spazio allocato vicino al blocco allocato oppure in una particolare tabella

Esempio:

```
char *s = new char[12];                 // Alloca un vettore di 12 char
.....
delete[] s;                             // Rilascia l'array
```

Eseguendo delete su un puntatore non const, il valore potrebbe essere azzerato

Esempio:

```
Cliente *array[10], *p;
int count = 0;
p = new Cliente;
array[count++] = p;
delete p;                               // Non portabile, dopo non potrebbe trovare "p"
for(int i = 0; i < count; i++)
        if(p == array[i]
                array[i] = 0;           // "p" potrebbe non essere mai trovato
                                        // Conviene eseguire delete p; a questo punto
```

Costruzione e distruzione degli oggetti

Un oggetto basato sullo stack viene automaticamente distrutto quando diventa non più visibile, mentre un oggetto creato sullo heap deve essere esplicitamente distrutto dal programmatore.
Nota: la memoria allocata sullo heap deve essere distrutta solo una volta; se si chiama new una volta per un singolo blocco e delete più di una volta, per lo stesso blocco si possono causare disastri.
Oggetti definiti sullo stack vengono costruiti nel momento in cui l'oggetto viene definito
Esempio:

```
int main() {
        Animale animale1;               // Esegue il costruttore in questo momento
        .........
        if(a == b) {
                Animale animale2;       // Esegue il costruttore solo se a == b
                .......
        }
int Animale animale3; }                 // Esegue il costruttore in questo momento
```

Gli oggetti sullo heap sono costruiti quando viene eseguito l'operatore new (una malloc non esegue il costruttore).
Esempio:

```
int main() {
        Animale *pt;
        .......
        pt = new Animale;               // Esegue il costruttore in questo momento
        .......
}
```

Oggetti definiti sullo stack vengono distrutti quando l'oggetto perde visibilità.
Esempio:

```
int main() {
        Animale animale1;
        ......
        if(a == b) {
                Animale animale2;
                .......
        }                               // Esegue il distruttore di animale2
        Animale animale3;
        .......
}                                       // Esegue il distruttore di animale1 e animale3
```

Oggetti sullo heap vengono distrutti quando viene eseguito l'operatore delete
Esempio:

```
int main() {
        Animale *pt;
        .......
        pt = new Animale;
        .......
        delete pt;                      // Esegue il distruttore in questo momento
}
```

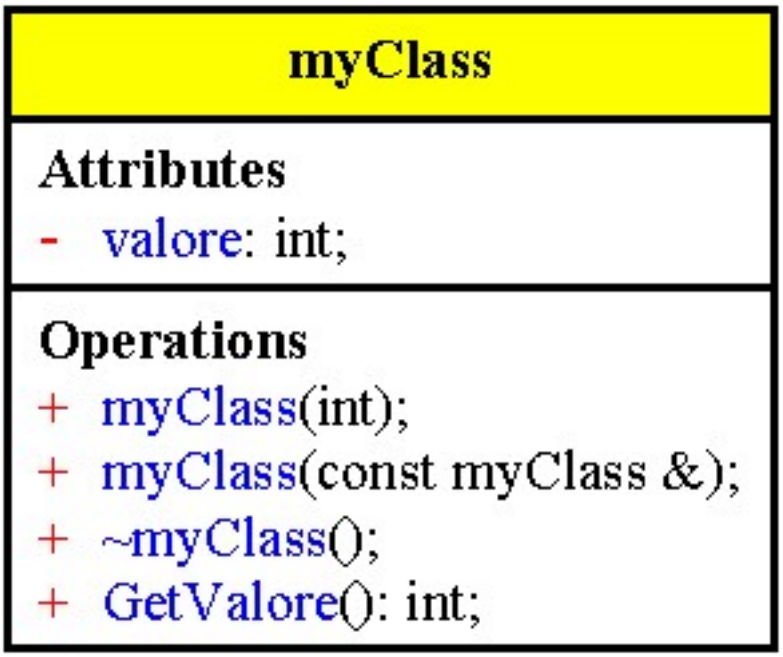
Ritorna

```
/* AStack.cpp
 * Allocazione di oggetti nello stack
 */
#pragma hdrstop
#include <condefs.h>
#include <iostream.h>
#include <fstream.h>
#include "myClass.h"
//---------------------------------------------------------------------------
USEUNIT("myClass.cpp");
//---------------------------------------------------------------------------
#pragma argsused
ofstream out("myClass.log");
int Funz(myClass);
void Attesa(char *);
int main(int argc, char* argv[]) {
  out << "In main, creazione di oggetto(3) ..." << endl;
  myClass oggetto(3);
  out << "In main, oggetto.valore = " << oggetto.GetValore() << endl;
  out << "In main, passaggio per valore di oggetto a Funz() ..." << endl;
  Funz(oggetto);
  out << "In main, uscita da Funz(). Fine ..." << endl;
  Attesa("terminare");
  return 0;
}
int Funz(myClass c) {
  out << "In Funz(), c.valore = " << c.GetValore() << endl;
  out << "Uscita da Funz() ..." << endl;
  return c.GetValore();
}
void Attesa(char * str) {
  cout << "Esaminare il file myClass.log";
  cout << "\n\n\tPremere return per " << str;
  cin.get();
}
/* Il passaggio per valore di "oggetto" a "Funz()" crea una copia
 * nel campo d'azione di "Funz()".
 * Quando tale copia esce dal campo di azione viene richiamato il distruttore
 */
```

```cpp
/* myClass.h
 */
//---------------------------------------------------------------------------
#ifndef myClassH
#define myClassH
#include <fstream.h>
extern ofstream out;
class myClass {
public:
   myClass(int val = 0);
   myClass(const myClass&);
   ~myClass();
   int GetValore() const {
      return valore;
   }
private:
   int valore;
};
#endif
```

```cpp
//---------------------------------------------------------------------------
/* myClass.cpp
 */
//---------------------------------------------------------------------------
#pragma hdrstop
#include "myClass.h"
//---------------------------------------------------------------------------
#pragma package(smart_init)
myClass::myClass(int val): valore(val) {
  out << "Costruttore di myClass" << endl;
}
myClass::myClass(const myClass & rhs): valore(rhs.valore) {
  out << "Costruttore di copia di myClass" << endl;
}
myClass::~myClass() {
  out << "Distruttore di myClass" << endl;
}
```

Diagramma U. M. L. delle classi



Ritorna

```
/* AHeap.cpp
 * Allocazione di oggeti nella heap
 */
#pragma hdrstop
#include <condefs.h>
#include <iostream.h>
#include <fstream.h>
#include "myClass.h"
//---------------------------------------------------------------------------
USEUNIT("myClass.cpp");
//---------------------------------------------------------------------------
#pragma argsused
ofstream out("myClass.log");
int Funz(myClass);
int Funz(myClass *);
void Attesa(char *);
int main(int argc, char* argv[]) {

  out << "In main, creazione di oggetto(3) nella heap Pobj-> ..." << endl;
  myClass * Pobj = new myClass(3);
  out << "In main, Pobj->valore = " << Pobj->GetValore() << endl;
  out << "In main, passaggio per valore di oggetto a Funz(*Pobj) ..." << endl;
  Funz(*Pobj);
  out << "In main, uscita da Funz(*Pobj)." << endl;
  out << "In main, passaggio per puntatore di oggetto a Funz(Pobj) ..." << endl;
  Funz(Pobj);
  out << "In main, uscita da Funz(Pobj)." << endl;
  out << "Cancellazione di Pobj ..." << endl;
  delete Pobj;
  out << "In main, Fine ..." << endl;
  Attesa("terminare");
  return 0;
}
int Funz(myClass c) {
  out << "In Funz(), c.valore = " << c.GetValore() << endl;
  out << "Uscita da Funz() ..." << endl;
  return c.GetValore();
}
int Funz(myClass *c) {
  out << "In Funz(*), c->valore = " << c->GetValore() << endl;
  out << "Uscita da Funz(*) ..." << endl;
  return c->GetValore();
}
void Attesa(char * str) {
  cout << "Esaminare il file myClass.log";
  cout << "\n\n\tPremere return per " << str;
  cin.get();
}
/* All'uscita dal main si deve richiamare esplicitamente delete
 * per distruggere l'oggetto costruito nella heap
 */
```

Ritorna

```
/* Newdel.cpp
 * Operatori new e delete
 */
#pragma hdrstop
#include <condefs.h>
#include <iostream.h>
//---------------------------------------------------------------------------
#pragma argsused
struct data {
  int mese;
  int giorno;
  int anno;
};
void Attesa(char *);
int main(void) {
  int index, *point1, *point2;

  point1  = &index;
  *point1 = 77;
  point2  = new int;
  *point2 = 173;
  cout << "I valori sono: " << index << "  "
       << *point1 << "  " << *point2 << "\n";
  point1 = new int;
  point2 = point1;
  *point1 = 999;
  cout << "I valori sono: " << index << "  "
       << *point1 << "  " << *point2 << endl;
  delete point1;

  float *float_point1, *float_point2 = new float;

  float_point1  = new float;
  *float_point2 = 3.14159;
  *float_point1 = 2.4 * (*float_point2);
  cout << "I valori sono: " << *float_point1 << "  "
       << *float_point1 << "  " << *float_point2 << endl;
  delete float_point2;
  delete float_point1;

  data *data_pt;

  data_pt        = new data;
  data_pt->mese = 10;
  data_pt->giorno   = 18;
  data_pt->anno  = 1938;
  cout << data_pt->giorno << "/"
       << data_pt->mese   << "/"
       << data_pt->anno   << endl;
  delete data_pt;

  char *c_point;

  c_point = new char[37];
  strcpy(c_point, "Frase iniziale");
  char * pc = c_point;
  cout << c_point << endl;
  c_point[0] = c_point[1] = '-';
  delete []c_point;
  cout << pc << endl;  // Non prevedibile
  c_point = new char[sizeof(data) + 133]; // Valido ma non consigliabile
  delete c_point; // rende disponibile solo un char
  Attesa("terminare");
  return 0;
}
void Attesa(char * str) {
  cout << "\n\n\tPremere return per " << str;
  cin.get();
}
```

Ritorna

```
/* AHeapArr.cpp
 * Allocazione di array oggeti nella heap
 */
#pragma hdrstop
#include <condefs.h>
#include <iostream.h>
#include <fstream.h>
#include "myClass.h"
//---------------------------------------------------------------------------
USEUNIT("myClass.cpp");
//---------------------------------------------------------------------------
#pragma argsused
ofstream out("myClass.log");
void  Funz(void);
void Attesa(char *);
int main(int argc, char* argv[]) {
  out << "In main, chiamata di Funz() ..." << endl;
  Funz();
  out << "In main, uscita da Funz()." << endl;
  Attesa("terminare");
  return 0;
}
void  Funz(void) {
  const int size = 5;
  myClass *PArray = new myClass[size];
  for(int i = 0; i < size; i++)
    PArray[i].SetValore(i);
  for(int j = 0; j < size; j++)
    out << "PArray[" << j << "] = " <<  PArray[j].GetValore() << endl;
  delete [] PArray;;
}
void Attesa(char * str) {
  cout << "Esaminare il file myClass.log";
  cout << "\n\n\tPremere return per " << str;
  cin.get();
}
/* All'uscita dal main si deve richiamare esplicitamente delete
 * per distruggere l'oggetto costruito nella heap
 */
```

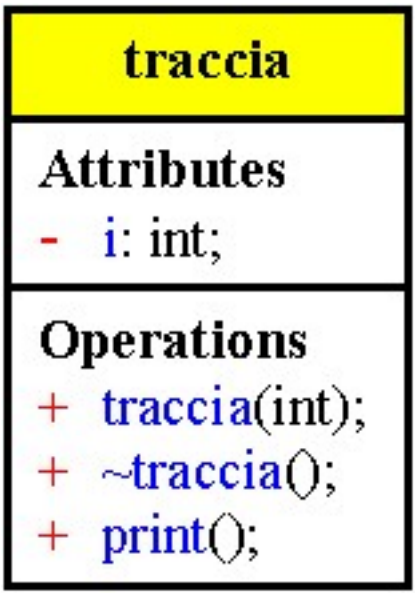
Ritorna

```
/* Freestor.cpp
 * Distruzione oggetti dinamici
 */
#pragma hdrstop
#include <condefs.h>
#include <iostream.h>
#include "Traccia.h"
//---------------------------------------------------------------------------
USEUNIT("Traccia.cpp");
//---------------------------------------------------------------------------
#pragma argsused
void g(void);
void f(void);
void Attesa(char *);
int main(int argc, char* argv[]) {
  g();
  // Il distruttore e' chiamato una sola volta, per l'oggetto locale!
  // Il puntatore "free" diventa non piu' visibile, ma l'oggetto al quale
  // punta e' ancora valido
  f();
  Attesa("terminare");
  return 0;
}
void Attesa(char * str) {
  cout << "\n\n\tPremere return per " << str;
  cin.get();
}
void g(void) {
  traccia local(10);        // Crea un oggetto locale
  traccia *free = new traccia(20);    // Memorizza l'oggetto
  local.print();            // Chiama una funzione membro per una variabile
  free->print();            // Chiama una funzione membro per un puntatore
}

void f(void) {
  traccia *free = new traccia(30);
  free->print();            // Chiama una funzione membro per un puntatore
  delete free;              // delete deve gestire l'indirizzo iniziale
                            // dell'oggetto creato con new.
}
/*
 * Mentre un oggetto basato sullo stack viene automaticamente distrutto quando
 * diventa non piu' visibile, un oggetto creato sullo heap deve essere
 * esplicitamente distrutto dal programmatore.
 * E' possibile che un oggetto non venga distrutto, e che il puntatore diventi
 * non piu' visibile (es. g()) e' importante distruggere l'oggetto prima di
 * perderne le tracce come in f()
 */
```

```
/* Traccia.h
 * Una classe per esaminare quando gli oggetti vengono creati e distrutti
 */
//---------------------------------------------------------------------------
#ifndef TracciaH
#define TracciaH
class traccia {
  int i;
public:
  traccia(int x = 0);
  ~traccia();
  void print();
};
#endif
```

```
//---------------------------------------------------------------------------
/* Traccia.cpp
 * Una classe per esaminare quando gli oggetti vengono creati e distrutti
 */
//---------------------------------------------------------------------------
#pragma hdrstop
#include <iostream.h>
#include "Traccia.h"
//---------------------------------------------------------------------------
#pragma package(smart_init)
traccia::traccia(int x) {
  cout << "Costruttore chiamato con argomento " << x << endl;
  i = x;
}
traccia::~traccia() {
  cout << "Distruttore chiamato per l'oggetto w/ i = " << i << endl;
}
void traccia::print() {
  cout << "traccia::print(); i = " << i << endl;
}
```

Diagramma U. M. L. delle classi



Ritorna

```
/* Fshandlr.cpp
 * Si definisce una propria funzione, da chiamare quando la
 * memoria si esaurisce
 */
#pragma hdrstop
#include <condefs.h>
#include <iostream.h>
#include <new.h>                // Dichiara set_new_handler()
//---------------------------------------------------------------------------
#pragma argsused
unsigned long cont = 0;
void memoria_esaurita(void);
void Attesa(char *);
int main(int argc, char* argv[]) {
  set_new_handler(memoria_esaurita);
  while(1) {
    cont++;
    new char;                   // Questo ciclo esaurisce la memoria libera!
  }
  Attesa("terminare");
  return 0;
}
void memoria_esaurita(void) {
  cout << "Memoria libera esaurita!  " << endl;
  cout << "Dopo  " << cont << "  Allocazioni" << endl;
  Attesa("terminare");
  exit(1);
}
void Attesa(char * str) {
  cout << "\n\n\tPremere return per " << str;
  cin.get();
}
/*
 * In questo caso set_new_handler() ha come argomento l'indirizzo di
 * memoria_esaurita() e la installa come funzione da chiamare quando la
 * memoria libera si esaurisce
 */
```

Steam di Input/Output

Le funzioni di I/O si occupano esclusivamente del processo di conversione degli oggetti dotati di tipo in sequenze di caratteri (o di bit) e viceversa.
Uno stream e' un oggetto che formatta e manipola bytes
La libreria iostream comprende due famiglie di classi una derivata dalla classe streambuf e l'altra derivata dalla classe ios.
La classe streambub fornisce una interfaccia con i device fisici e i metodi per bafferizzare e manipolare gli stream.
La classe ios contiene un puntatore a uno streambuf e gestisce la formattazione dell'I/O

File iostream

Per gestire i file occorre includere il file header fstream.h e creare e utilizzare gli oggetti fstream.
Un oggetto ifstream è un tipo speciale di iostream che apre e legge un file.
Un oggetto ofstream è un tipo speciale di iostream che apre e scrive un file.
I file vengono chiusi automaticamente al momento della distruzione degli oggetti.

Modalita' di apertura

ios::in	Apre un input file (utile per ofstream per evitare di troncare un file esistente)
ios::out	Apre un output file
ios::app	Apre un output file per aggiungere in fondo
ios:ate	Apre un file esistente e si posiziona al fondo
ios::nocreate	Apre un file solo se esiste
ios::noreplace	Apre un file solo se non esiste
ios::trunc	Apre un file e cancella il file se esiste gia'
ios::binary	Apre un file in formato binario

Posizionamento nell'iostream

ios::beg	All'inizio dello stream
ios::cur	Nella posizione corrente dello stream
ios::end	Alla fine dello stream

Stato dell'I/O

Quando si aprono i file o si eseguono operazioni di I/O si devono controllare se le operazioni hano avuto successo, a tale scopo è possibile utilizzare le funzioni membro della classe ios

iso::rdstate()	Restituisce il valore della variabile state	
	ios::goodbit	Operazione eseguita con successo
	ios::eofbit	Fine del file
	ios::failbit	Errore nell'ultima operazione di I/O
	ios::badbit	Tentativo di eseguire un'operazione errata
iso::eof()	Vero alla fine del file	
ios::fail()	Vero se l'operazione non è andata a buon fine	
ios::bad()	Vero se si è cercato di eseguire una operazione errata	
ios::good()	Vero quando l'ultima operazione ha avuto successo	

Formattazione dell'output

La classe ios contiene membri dati che memorizzano la formattazione dei dati riguardanti lo stream
La formattazione puo' essere controllata attraverso le funzioni membro e i manipolatori
La classe ios contiene dei dati membro che memorizzano lo stato della formattazione dei dati riguardanti lo stream; alcuni comprendono un range di dati gli altri sono dei flag

ios::flags() modifica tutti i flag
ios::setf() imposta i flag
ios::unsetf() azzera i flag

Abbiamo due tipi di flag:

on / off
uno flag impostato tra un gruppo di valori possibili

ios::skipws	Non considera gli spazi (default per l'input)
ios::showbase	Indica la base dei numeri
ios::showpoint	Visualizza il punto decimale per i float
ios::uppercase	Visualizza in maiuscolo i caratteri A-F, E dei numeri
ios::showpos	Visualizza in + per i valori positivi
ios::unitbuf	Lo stream e' scaricato ad ogni inserzione
ios::stdio	Sincronizza lo stream con I/O standard del C

ios::basefield

ios::dec	Formato base decimale
ios::hex	Formato base esadecimale
ios::oct	Formato base ottale

ios::floatfield

ios::scientific	Visualizza in formato scientifico, la precisione indica le cifre dopo il punto decimale
ios::fixed	Visualizza in formato fisso, la precisione indica le cifre dopo il punto decimale
bit non settati	La precisione indica il totale delle cifre significative

ios::adjustfield

ios::left	Allineamento a sinistra, riempimento a destra
ios::right	Allineamento a destra, riempimento a sinistra
ios::internal	Aggiunge i caratteri di riempimento dopo il segno e la base ma prima del valore

Funzioni

int ios::width()	Legge la larghezza corrente (usato per l'inserzione e l'estrazione)
int ios::width(int n)	Imposta la larghezza e restituisce la larghezza precedente
int ios::fill()	Legge in carattere di riempimento corrente
int ios:fill(int n)	Imposta il carattere di riempimento e restituisce il precedente
int ios::precision()	Legge la precisone dei numeri float (default 6)
int ios::precision(int n)	Imposta la precisione dei float e restituisce la precisione precedente

Manipolatori

I manipolatori di iostream sono operatori che cambiano lo stato dello stream di output

showbase / noshowbase	Visualizza la base dei numeri
showpos / noshowpos	Visualizza il segno + dei numeri positivi
uppercase / nouppercase	Visualizza in maiuscolo i caratteri A-F, E dei numeri
showpoint / noshowpoint	Visualizza il punto decimale per i float
skipws / noskipws	Non considera gli spazi (default per l'input)
left / right / internal	Tipo di allineamento
scientific / fixed	Formato dei float

Manipolatori con argomenti

setioflags(n)	Imposta i flag specificati da n
resetioflags(n)	Azzera i flag specificati da n
setbase(n)	Cambia la base a n (10, 8, 16)
setfill(c)	Cambia il carattere di riempimento a c
setprecision(n)	Cambia la precisione a n
setw(n)	Cambia la larghezza del campo a n

Funzioni membro

La classe iostream contiene molte funzioni membro
Get() ha lo scopo di ottenere un singolo carattere (o una stringa) dall'oggetto cin
Get() ritorna un valore non-zero se vi sono dei caratteri disponibili in input
Put() invia caratteri all'oggetto cout
Per poter leggere e usare più di un carattere alla volta dallo standard input è necessario utilizzare un buffer, cioè un'area di memoria che contiene un insieme di dati dello stesso tipo.
La funzione get() aggiunge alla fine della stringa il byte zero che rappresenta il carattere di terminazione delle stringhe.
Per default il carattere di terminazione è il carattere di ritorno a capo, per cambiare il delimitatore di stringa occorre passare a get() come terzo argomento il nuovo carattere di terminazione racchiuso da doppi apici.
Il carattere di terminazione deve essere letto ed eliminato, altrimenti la prossima volta che get() viene chiamata legge per prima cosa il carattere di terminazione e si ferma.

Ritorna

```
/* Fstream.cpp
 * Utilizzo dei file
 */
#pragma hdrstop
#include <condefs.h>
#include <iostream.h>
#include <fstream.h>
#include <ios.h>
#include <process.h>
//---------------------------------------------------------------------------
#pragma argsused
void Attesa(char *);
int main(int argc, char* argv[]) {
ifstream infile;
ofstream outfile;
ofstream printer;
char filename[20];
  cout << "Indica il file da copiare ----> ";
  cin >> filename;
  infile.open(filename);
  if (!infile) {
    cout << "Il file di imput non puo' essere aperto.\n";
    Attesa("terminare");
    exit(1);
  }
  outfile.open("copy");
  if (!outfile) {
    cout << "Il file di output non puo' essere aperto.\n";
    Attesa("terminare");
    exit(1);
  }
/*
  printer.open("PRN");
  if (!printer) {
    cout << "Problemi con la stampante.\n";
    Attesa("terminare");
    exit(1);
  }
*/
  cout << "Tutti i file sono stati aperti.\n";
  char one_char;
//  printer << "Inizio della copia.\n\n";
  cout << "Inizio della copia.\n\n";
  while (infile.get(one_char)) {
    outfile.put(one_char);
//    printer.put(one_char);
  }
//  printer << "\n\nFine della copia.\n";
  cout << "Fine della copia.\n";
  infile.close();
  outfile.close();
//  printer.close();
  Attesa("terminare");
  return 0;
}
void Attesa(char * str) {
  cout << "\n\n\tPremere return per " << str;
  cin.ignore(4, '\n');
  cin.get();
}
```

Ritorna

```
/* Strfile.cpp
 * Stream I/O con file  Differenza tra get() e getline()
 */

#pragma hdrstop
#include <condefs.h>
#include <iostream.h>
#include <fstream.h>
#define SZ 100
//---------------------------------------------------------------------------
#pragma argsused
void Attesa(char *);
int main(int argc, char* argv[]) {
  char buf[SZ];
  ifstream in("strfile.cpp");      // Lettura
  if(!in) {
    cout << "Errore apertura strfile.cpp" << endl;
    Attesa("terminare");
    exit(1);
  }
  ofstream out("strfile.out");     // Scrittura
  if(!out) {
    cout << "Errore apertura strfile.out" << endl;
    Attesa("terminare");
    exit(1);
  }
  int i = 1;                       // Contatore di linee
  // Un approccio poco conveniente per fare l'input di una linea
  while(in.get(buf, SZ)) {         // Lascia '\n' in input
    in.get();                      // Trova il prossimo carattere
    out << i++ << ": " << buf << endl;
  }
  in.close();
  out.close();
  in.open("strfile.out");
  if(!in) {
    cout << "Errore apertura strfile.out" << endl;
    Attesa("terminare");
    exit(1);
  }
  // Un  modo piu' conveniente per l'input di una linea
  while(in.getline(buf, SZ)) {     // Rimuove '\n'
    char *cp = buf;
    while(*cp != ':')
      cp++;
    cp += 2;                       // Esclude ": "
    cout << cp << endl;
  }
  Attesa("terminare");
  return 0;
}
void Attesa(char * str) {
  cout << "\n\n\tPremere return per " << str;
  cin.get();
}
```

Ritorna

```
/* Format.cpp
 * Funzioni di formattazione dell'output
 */

#pragma hdrstop
#include <condefs.h>
#include <ios.h>
#include <iostream.h>
#include <fstream.h>
//---------------------------------------------------------------------------
#pragma argsused
#define D(a) T << #a << endl; a
ofstream T("format.out");
void Attesa(char *);
int main(int argc, char* argv[]) {
  D(int i = 47;)
  D(float f = 2300114.414159;)
  char* s = "Una frase qualsiasi";
  D(T.setf(ios::unitbuf);) // Lo stream e' svuotato dopo ogni inserimento
//  D(T.setf(ios::stdio);) Sincronizza lo stream con il sistema C standard I/O
  D(T.setf(ios::showbase);) // Visualizza l'indicatore della base
  D(T.setf(ios::uppercase);)// Visualizza con lettere maiuscole (A-F,E) i numeri
  D(T.setf(ios::showpos);)  // Mostra il segno + nei valori positivi
  D(T << i << endl;)        // Il default e' dec
  D(T.setf(ios::hex, ios::basefield);)
  D(T << i << endl;)
  D(T.unsetf(ios::uppercase);)
  D(T.setf(ios::oct, ios::basefield);)
  D(T << i << endl;)
  D(T.unsetf(ios::showbase);)
  D(T.setf(ios::dec, ios::basefield);)
  D(T.setf(ios::left, ios::adjustfield);)
  D(T.fill('0');)
  D(T << "Carattere di riempimento: " << T.fill() << endl;)
  D(T.width(10);)  T << i << endl;
  D(T.setf(ios::right, ios::adjustfield);)
  D(T.width(10);)  T << i << endl;
  D(T.setf(ios::internal, ios::adjustfield);)
  D(T.width(10);)  T << i << endl;
  D(T << i << endl;)                          // Senza width(10)
  D(T.unsetf(ios::showpos);)
  D(T.setf(ios::showpoint);)
  D(T << "prec = " << T.precision() << endl;)
  D(T.setf(ios::scientific, ios::floatfield);)
  D(T << endl << f << endl;)
  D(T.setf(ios::fixed, ios::floatfield);)
  D(T << f << endl;)
  D(T.setf(0, ios::floatfield);)              // Automatico
  D(T << f << endl;)
  D(T.precision(20);)
  D(T << "prec = " << T.precision() << endl;)
  D(T << endl << f << endl;)
  D(T.setf(ios::scientific, ios::floatfield);)
  D(T << endl << f << endl;)
  D(T.setf(ios::fixed, ios::floatfield);)
  D(T << f << endl;)
  D(T.setf(0, ios::floatfield);)              // Automatico
  D(T << f << endl;)
  D(T.width(10);)  T << s << endl;
  D(T.width(40);)  T << s << endl;
  D(T.setf(ios::left, ios::adjustfield);)
  D(T.width(40);)  T << s << endl;
  D(T.unsetf(ios::showpoint);)
  D(T.unsetf(ios::unitbuf);)
//  D(T.unsetf(ios::stdio);)
  Attesa("terminare");
  return 0;
}
void Attesa(char * str) {
  cout << "\n\n\tPremere return per " << str;
  cin.get();
}
```

Ritorna

```cpp
/* Manips.cpp
 * Formattazione utilizzando i manipolatori
 */
#pragma hdrstop
#include <condefs.h>
#include <iostream.h>
#include <fstream.h>
#include <iomanip.h>
//---------------------------------------------------------------------------
#pragma argsused
void Attesa(char *);
int main(int argc, char* argv[]) {
  ofstream T("Traccia.out");
  int i = 47;
  float f = 2300114.414159;
  char* s = "Una frase qualsiasi";

  // setiosflags(n) Imposta i flags di formato secondo il valore di n
 T << setiosflags(ios::unitbuf | ios::showbase | ios::uppercase | ios::showpos);
  // unitbuf   Lo stream e' scaricato dopo ogni inserimento
  // stdio     Sincronizza lo stream con il sistema I/O del C
  // showbase  Indica la base dei numeri
  // uppercase Indica in maiuscolo le lettere dei numeri
  // showpos   Visualizza il segno + dei numeri positivi
  T << i << endl;                  // Default  dec
  T << hex << i << endl;
  // resetiosflags(n)  Azzera il flag di formati specificati da n
  T << resetiosflags(ios::uppercase) << oct << i << endl;
  // setf() Imposta il flag
  T.setf(ios::left, ios::adjustfield);
  T << resetiosflags(ios::showbase) << dec << setfill('0');
  // setfill(c) Imposta il carattere di riempimento a c
  T << "Carattere di riempimento: " << T.fill() << endl;
  // setw()    Modifica la larghezza del campo
  T << setw(10) << i << endl;
  T.setf(ios::right, ios::adjustfield);
  T << setw(10) << i << endl;
  T.setf(ios::internal, ios::adjustfield);
  T << setw(10) << i << endl;
  T << i << endl;                          // Senza setw(10)
  T << resetiosflags(ios::showpos)
    << setiosflags(ios::showpoint)
    << "prec = " << T.precision() << endl;
  T.setf(ios::scientific, ios::floatfield);
  T << f << endl;
  T.setf(ios::fixed, ios::floatfield);
  T << f << endl;
  T.setf(0, ios::floatfield);              // Automatico
  T << f << endl;
  T << setprecision(20);
  T << "prec = " << T.precision() << endl;
  T << f << endl;
  T.setf(ios::scientific, ios::floatfield);
  T << f << endl;
  T.setf(ios::fixed, ios::floatfield);
  T << f << endl;
  T.setf(0, ios::floatfield);              // Automatico
  T << f << endl;
  T << setw(10) << s << endl;
  T << setw(40) << s << endl;
  T.setf(ios::left, ios::adjustfield);
  T << setw(40) << s << endl;
  T << resetiosflags(ios::showpoint | ios::unitbuf );
  Attesa("terminare");
  return 0;
}
void Attesa(char * str) {
  cout << "\n\n\tPremere return per " << str;
  cin.get();
}
```

Ritorna

```
/* Iofile.cpp
 * Lettura e scrittura in un file
 */
#pragma hdrstop
#include <condefs.h>
#include <iostream.h>
#include <fstream.h>
//---------------------------------------------------------------------------
#pragma argsused
void Attesa(char *);
int main(int argc, char* argv[]) {
  ifstream in("Iofile.cpp");
  ofstream out("Iofile.out");
  out << in.rdbuf();               // Copia del file
  in.close();
  out.close();
  // Apre il file per la lettura e la scrittura:
  ifstream in2("iofile.out",ios::in|ios::out);
  ostream out2(in2.rdbuf());
  cout << in2.rdbuf();            // Visualizza tutto il file
  out2 << "Dove verra' scritto questo?";
  out2.seekp(0, ios::beg);
  out2 << "E dove avverra' questo?";
  in2.seekg(0, ios::beg);
  cout << in2.rdbuf();
  Attesa("terminare");
  return 0;
}
void Attesa(char * str) {
  cout << "\n\n\tPremere return per " << str;
  cin.get();
}
/* Tutto il testo inserito sovrascrive il testo esistente
 */
```

Ritorna

```cpp
/* Seeking.cpp
 * Posizionamento in iostreams
 */
#pragma hdrstop
#include <condefs.h>
#include <iostream.h>
#include <fstream.h>
//---------------------------------------------------------------------------
#pragma argsused
void Attesa(char *);
int main(int argc, char* argv[]) {
  if(argc < 2) {
     cout << "Introdurre come argomento il nome del file" << endl;
     Attesa("terminare");
     exit(1);
  }
  ifstream in(argv[1]);
  if(!in) {
     cout << "Errore apertura del file " << argv[1] << endl;
     Attesa("terminare");
     exit(1);
  }
  in.seekg(0, ios::end);        // Fine del file
  streampos sp = in.tellg();    // Dimensione del file
  cout << "Dimensione del file = " << sp << endl;
  in.seekg(-sp/10, ios::end);
  streampos sp2 = in.tellg();
  in.seekg(0, ios::beg);        // Inizio del file
  cout << in.rdbuf();           // Visualizza tutto il file
  in.seekg(sp2);                // Si posiziona in streampos
  // Visualizza última parte (1/10) del file:
  cout << endl << endl << in.rdbuf() << endl;
  Attesa("terminare");
  return 0;
}
void Attesa(char * str) {
  cout << "\n\n\tPremere return per " << str;
  cin.get();
}
/* streampos   posizione assoluta nello stream
 * tellp(), tellg() restituisce la posizione nello stream
 * tellp() per un ostream   "put pointer"
 * tellg() per un istream   "get pointer"
 * seekp(), seekg() si posiziona nello stream
 * seekp() per un ostream
 * seekg() per un istream
 * ios::beg Dalla posizione iniziale
 * ios::cur Posizione corrente
 * ios::end Dalla posizione finale
 */
```

Ritorna

```
/* Istring.cpp
 * input strstream
 */
#pragma hdrstop
#include <condefs.h>
#include <iostream.h>
#include <strstrea.h>
//---------------------------------------------------------------------------
#pragma argsused
void Attesa(char *);
int main(int argc, char* argv[]) {
  istrstream s("1.414 47 Questo e` un test");
  int i;
  float f;
  s >> i >> f;  // Lo spazio delimita l'input
  char buf[100];
  s >> buf;
  cout << "i = "   << i   << endl
       << "f = "   << f   << endl
       << "buf = " << buf << endl;
  cout << s.rdbuf();   // Legge la parte restante
  Attesa("terminare");
  return 0;
}
void Attesa(char * str) {
  cout << "\n\n\tPremere return per " << str;
  cin.get();
}
```

Ritorna

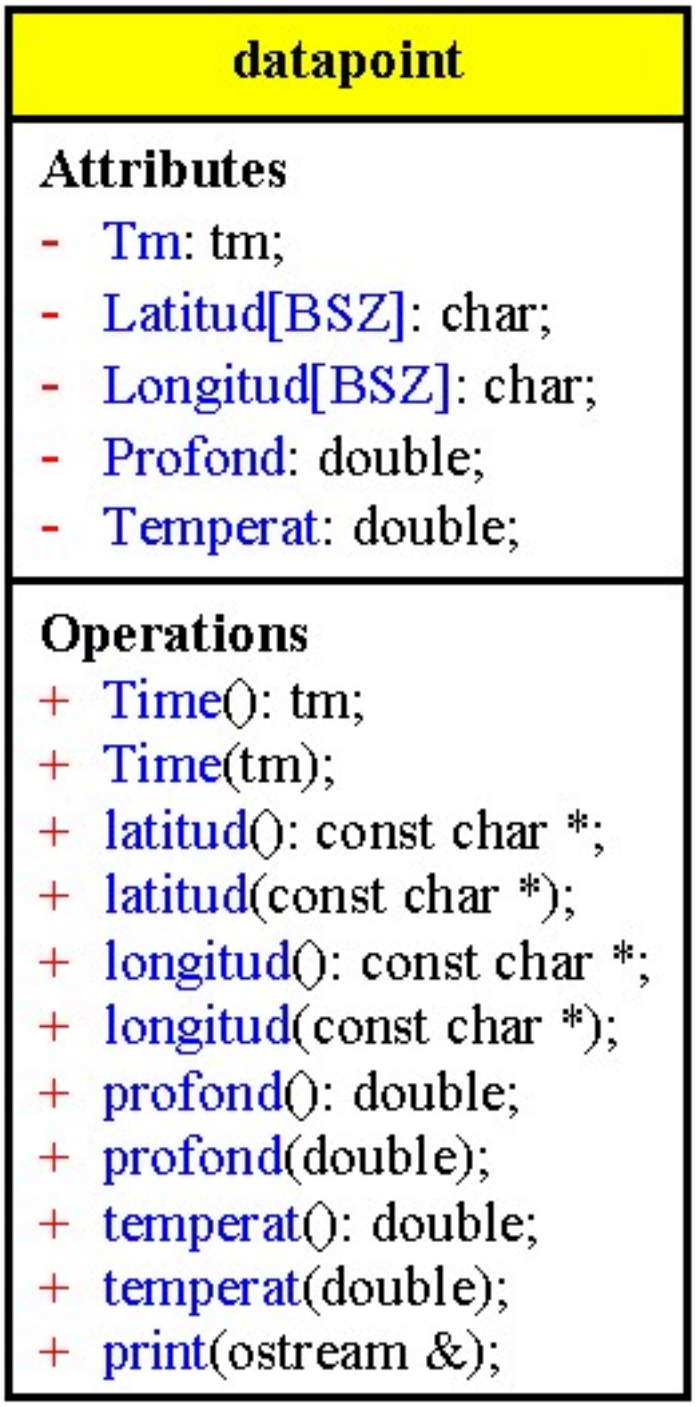
```
/* Ostring.cpp
 * output su strstream
 */
#pragma hdrstop
#include <condefs.h>
#include <iostream.h>
#include <strstrea.h>
#define SZ 100
//---------------------------------------------------------------------------
#pragma argsused
void Attesa(char *);
int main(int argc, char* argv[]) {
  cout << "Introdurre un intero, un float e una stringa:  ";
  int i;
  float f;
  cin >> i >> f;
  cin >> ws;    // Elimina lo spazio
  char buf[SZ];
  cin.getline(buf, SZ);   // Legge il resto della linea
  ostrstream os(buf, SZ, ios::app);
  os << endl;
  os << "intero = " << i << endl;
  os << "float  = " << f << endl;
  os << ends;
  cout << buf;
  cout << os.rdbuf();   // Lo stesso effetto
  cout << os.rdbuf();   // Non lo stesso effetto (il puntatore e`alla fine
  Attesa("terminare");
  return 0;
}
void Attesa(char * str) {
  cout << "\n\n\tPremere return per " << str;
  cin.get();
}
```

Ritorna

```cpp
/* datagen.cpp
 * Test generazione di dati
 */
#pragma hdrstop
#include <condefs.h>
#include <iostream.h>
#include <fstream.h>
#include "Datalog.h"
//---------------------------------------------------------------------------
USEUNIT("Datalog.cpp");
//---------------------------------------------------------------------------
#pragma argsused
void Attesa(char *);
int main(int argc, char* argv[]) {
  ofstream dati("dati.txt");
  ofstream datibin("dati.bin", ios::binary);
  time_t tempo = time(NULL);
  srand((unsigned)tempo);
  for(int i=0; i < 100; i++) {
    datapoint d;
    d.Time(*localtime(&tempo));
    tempo += 55;
    d.latitud("45*20'31\"");
    d.longitud("22*34'18\"");
    double nuovaprof = rand() % 200;
    double rapport = rand() % 100 + 1;
    nuovaprof += double(1) / rapport;
    d.profond(nuovaprof);
    double nuovatemp = 150 + rand()%200;
    rapport = rand()%100+1;
    nuovatemp += double(1) / rapport;
    d.temperat(nuovatemp);
    d.print(dati);
    datibin.write((unsigned char*)&d, sizeof(d));
  }
  Attesa("terminare");
  return 0;
}
void Attesa(char * str) {
  cout << "\n\n\tPremere return per " << str;
  cin.get();
}
```

```
/* Datalog.h
 */
//---------------------------------------------------------------------------
#ifndef DatalogH
#define DatalogH
#include <time.h>
#include <iostream.h>
#define BSZ 10
class datapoint {
  tm Tm;                    // ora e giorno
  char Latitud[BSZ];
  char Longitud[BSZ];
  double Profond;
  double Temperat;
public:
  tm Time();                // Legge il tempo
  void Time(tm T);          // Imposta il tempo
  const char *latitud();
  void latitud(const char *l);
  const char *longitud();
  void longitud(const char *l);
  double profond();
  void profond(double d);
  double temperat();
  void temperat(double t);
  void print(ostream &os);
};
//---------------------------------------------------------------------------
#endif
```

Diagramma U. M. L. delle classi



```
/* Datalog.cpp
 */
//---------------------------------------------------------------------------
#pragma hdrstop
#include <iomanip.h>
#include "Datalog.h"
//---------------------------------------------------------------------------
#pragma package(smart_init)
tm datapoint::Time() {
  return Tm;
}
void datapoint::Time(tm T) {
  Tm = T;
}
const char *datapoint::latitud() {
  return Latitud;
}
void datapoint::latitud(const char *l) {
  Latitud[BSZ -1] = 0;
  strncpy(Latitud, l, BSZ-1);
}
const char *datapoint::longitud() {
  return Longitud;
}
void datapoint::longitud(const char *l) {
  Longitud[BSZ-1] = 0;
  strncpy(Longitud, l, BSZ-1);
}
double datapoint::profond() {
  return Profond;
}
void datapoint::profond(double d) {
  Profond = d;
}
double datapoint::temperat() {
  return Temperat;
}
void datapoint::temperat(double t) {
  Temperat = t;
}
void datapoint::print(ostream &os) {
  os.setf(ios::fixed, ios::floatfield);
  os.precision(4);
  os.fill('0');
  os << setw(2) << Time().tm_mon  << '\\'
     << setw(2) << Time().tm_mday << '\\'
     << setw(2) << Time().tm_year << ' '
     << setw(2) << Time().tm_hour << ':'
     << setw(2) << Time().tm_min  << ':'
     << setw(2) << Time().tm_sec;
  os.fill(' ');
  os << "  Lat:"  << setw(9) << latitud()
     << ", Long:" << setw(9) << longitud()
     << ", Prof:" << setw(9) << profond()
     << ", Temp:" << setw(9) << temperat()
     << endl;
}
```

```
/* Datascan.cpp
 * Esamina la validita` dei dati memorizzati in binario
 * dal programma datagen.cpp
 */
#pragma hdrstop
#include <condefs.h>
#include <iostream.h>
#include <fstream.h>
#include <iomanip.h>
#include <strstrea.h>
#include "Datalog.h"
//---------------------------------------------------------------------------
USEUNIT("Datalog.cpp");
//---------------------------------------------------------------------------
#pragma argsused
void Attesa(char *);
int main(int argc, char* argv[]) {
  ifstream datibin("dati.bin", ios::binary);
  if(!datibin) {
     cout << "Errore apertura dati.bin" << endl;
     Attesa("terminare");
     exit(1);
  }
  ofstream verif("dati2.txt");
  datapoint d;
  while(datibin.read((unsigned char *)&d, sizeof(d)))
    d.print(verif);
  datibin.clear();
  int numrec = 0;
  cout.setf(ios::left, ios::adjustfield);
  cout.setf(ios::fixed, ios::floatfield);
  cout.precision(4);
  for(;;) {
    datibin.seekg(numrec * sizeof(d), ios::beg);
    cout << "record " << numrec << endl;
    if(datibin.read((unsigned char *)&d, sizeof(d))) {
      cout << asctime(&(d.Time()));
      cout << setw(15) << "Latitudine"
           << setw(15) << "Longitudine"
           << setw(15) << "Profondita`"
           << setw(15) << "Temperatura"
           << endl;
      cout << setfill('-') << setw(60) << '-'
           << setfill(' ') << endl;
      cout << setw(15) << d.latitud()
           << setw(15) << d.longitud()
           << setw(15) << d.profond()
           << setw(15) << d.temperat()
           << endl;
    } else {
      cout << "Numero record non valido" << endl;
      datibin.clear();
    }
    cout << endl
         << "Introdurre il numero del record, x per quit: ";
    char buf[10];
    cin.getline(buf, 10);
    if(buf[0] == 'x')
      break;
    istrstream input(buf, 10);
    input >> numrec;
  }
  Attesa("terminare");
  return 0;
}
void Attesa(char * str) {
  cout << "\n\n\tPremere return per " << str;
  cin.ignore(4, '\n');
  cin.get();
}
```

Costruttore di copia

Quando si passano i parametri a una funzione per valore o una funzione ritorna un valore il C++ esegue delle operazioni nascoste.
Per quanto riguarda gli oggetti predefiniti (int, float,..) il compilatore li conosce e possiede le regole per passare o ritornare valori, per i tipi definiti dall'utente non ne conosce le regole.
Quando l'utente non interviene il compilatore esegue una copia degli elementi della struttura e chiama i costruttori per gli oggetti membri presenti.
Ciò funziona per oggetti semplici ma non ha l'effetto desiderato quando l'oggetto contiene puntatori o membri di altri oggetti.
Il C++ permette di acquisire il controllo delle attività nascoste definendo il costruttore di copia che viene automaticamente chiamato per gestire il passaggio di argomenti e il ritorno dei valori.
Un costruttore di copia è un costruttore che accetta come argomento un riferimento a un oggetto della propria classe.
X(X&)
Il costruttore di copia deve eseguire tutte le inizializzazioni necessarie quando un oggetto viene creato a partire da un altro
Il costruttore di copia viene usato in tre casi:

- Inizializzazione esplicita di un oggetto con un altro della stessa classe.
- Il passaggio di un oggetto per valore come parametro di una funzione.
- Passaggio di un oggetto come valore di ritorno di una funzione.

Di solito si deve sostituire il costruttore di copia di default solo per una classe che ha dei puntatori come membri, in quanto sono copiati così come sono indipendentemente dal loro significato.
Una soluzione è di allocare un nuovo puntatore e quindi copiare il valore del puntatore invece del suo indirizzo.
Questa soluzione spreca memoria, ogni oggetto puntato deve essere replicato, e a volte serve riferirsi allo stesso oggetto dall'interno di due più oggetti differenti.
Possibili soluzioni sono:

- fare una copia dell'oggetto puntato solo quando questo cambia
- implementare un meccanismo di conteggio dei puntatori di modo che il suo distruttore rilascia la memoria solo quando non ci sono più riferimenti ad esso.

Creazione di oggetti temporanei

Quando il compilatore trova un costrutto del tipo: un nome di una classe con una lista di argomenti, lo tratta come una chiamata a un costruttore per un oggetto temporaneo, cioé un oggetto senza nome e normalmente di breve durata.
Esempio: fp(point(1.1, 2.2));

si crea un oggetto temporaneo per passarlo come argomento alla funzione

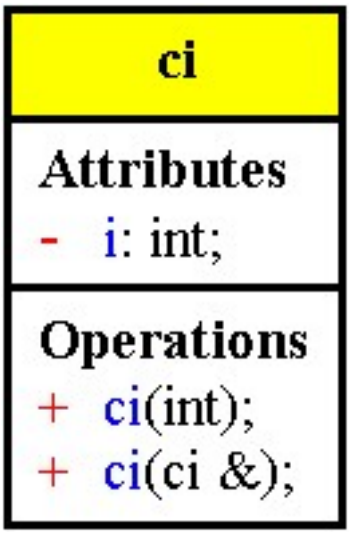
Esempio: return point(new_x, new_y);

viene allocato all'interno della funzione lo spazio per contenere un oggetto point,
viene chiamato il costruttore point::point(float, float) pe rinizializzare l'oggetto
l'oggetto point temporaneo viene copiato all'esterno della funzione pe rmezzo della istruzione return
l'oggetto temporaneo diventa non più visibile

Ritorna

```
/* Ci.h
 */
//---------------------------------------------------------------------------
#ifndef CiH
#define CiH
class ci {
  int i;
public:
  ci(int);
  // Inizializzatore di copia
  ci(ci &);
};
//---------------------------------------------------------------------------
#endif

//---------------------------------------------------------------------------
/* Ci.cpp
 */
//---------------------------------------------------------------------------
#pragma hdrstop
#include <iostream.h>
#include "Ci.h"
//---------------------------------------------------------------------------
#pragma package(smart_init)
ci::ci(int j) {
 i = j;
}
ci::ci(ci & rv) {
  cout << "Inizializzatore di copia chiamato" << endl;
  i = rv.i;             // Copia in rvalue
}

//---------------------------------------------------------------------------
/* MCi.cpp
 */
#pragma hdrstop
#include <condefs.h>
#include <iostream.h>
#include "Ci.h"
//---------------------------------------------------------------------------
USEUNIT("Ci.cpp");
//---------------------------------------------------------------------------
#pragma argsused
void Attesa(char *);
int main(int argc, char* argv[]) {
  ci original(1);
  ci copy1(original);   // Inizializzatore di copia chiamato
  ci copy2 = original; // anche qui
  Attesa("terminare");
  return 0;
}
void Attesa(char * str) {
  cout << "\n\n\tPremere return per " << str;
  cin.get();
}
/*
 * Se la copia automatica e` inaproppriata e/o occorrono operazioni di
 * inizializzazione di cancellazione quando le variabili sono passate
 * alla funzione, o quando vengono ritornate dalla funzione, il C++
 * permette di definire una funzione per eseguire il passaggio di argomenti
 * e il ritorno di valori, questa funzione e` chiamata costruttore di copia.
 */
```

Diagramma U. M. L. delle classi

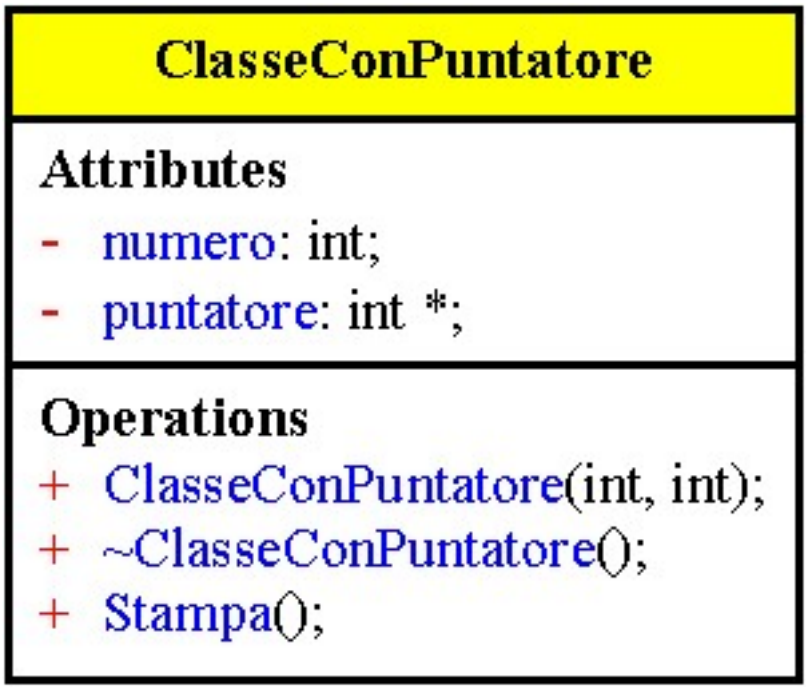


Ritorna

```
/* Pointcl1.cpp
 * classe con puntatori che illustra la necessita' di un costruttore di
 * copia per evitare dangling references
 */
#pragma hdrstop
#include <condefs.h>
#include <iostream.h>
#include "ClasseP.h"
//---------------------------------------------------------------------------
USEUNIT("ClasseP.cpp");
//---------------------------------------------------------------------------
#pragma argsused
void Attesa(char *);
int main(int argc, char* argv[]) {
  cout << "Sto creando l'oggetto A" << endl;
  ClasseConPuntatore oggA (5, 8);
  {
    cout << endl;
    cout << "Sto creando l'oggetto B" << endl;
    ClasseConPuntatore oggB = oggA;
    oggB.Stampa();
    cout << endl;
    cout << "L'oggetto B sta per essere distrutto" << endl;
  }
  cout << endl;
  cout << "L'oggetto A ha un puntatore dangling" << endl;
  oggA.Stampa();
  cout << endl;
  cout << "L'oggetto A sta per essere distrutto" << endl;
  Attesa("terminare");
  return 0;
}
void Attesa(char * str) {
  cout << "\n\n\tPremere return per " << str;
  cin.get();
}
```

```
/* ClasseP.h
 */
//---------------------------------------------------------------------------
#ifndef ClassePH
#define ClassePH
class ClasseConPuntatore {
  int   numero;
  int*  puntatore;
public:
  // costruttore
  ClasseConPuntatore (int, int);
  // distruttore
  ~ClasseConPuntatore();
  void Stampa();
};
//---------------------------------------------------------------------------
#endif

//---------------------------------------------------------------------------
/* ClasseP.cpp
 */
//---------------------------------------------------------------------------
#pragma hdrstop
#include <iostream.h>
#include "ClasseP.h"
//---------------------------------------------------------------------------
#pragma package(smart_init)
ClasseConPuntatore::ClasseConPuntatore (int a, int b) {
  numero = a;
  puntatore = new int;
  *puntatore = b;
  cout << "Nuovo puntatore - indirizzo: " << puntatore
       << ", valore:" << *puntatore << endl;
}
ClasseConPuntatore::~ClasseConPuntatore() {
  cout << "Distruzione puntatore - indirizzo: " << puntatore
       << ", valore:" << *puntatore << endl;
  delete puntatore;
}
void ClasseConPuntatore::Stampa() {
  cout << "Il valore puntato e' " << *puntatore
       << " e il suo indirizzo e' " << puntatore << endl;
}
```

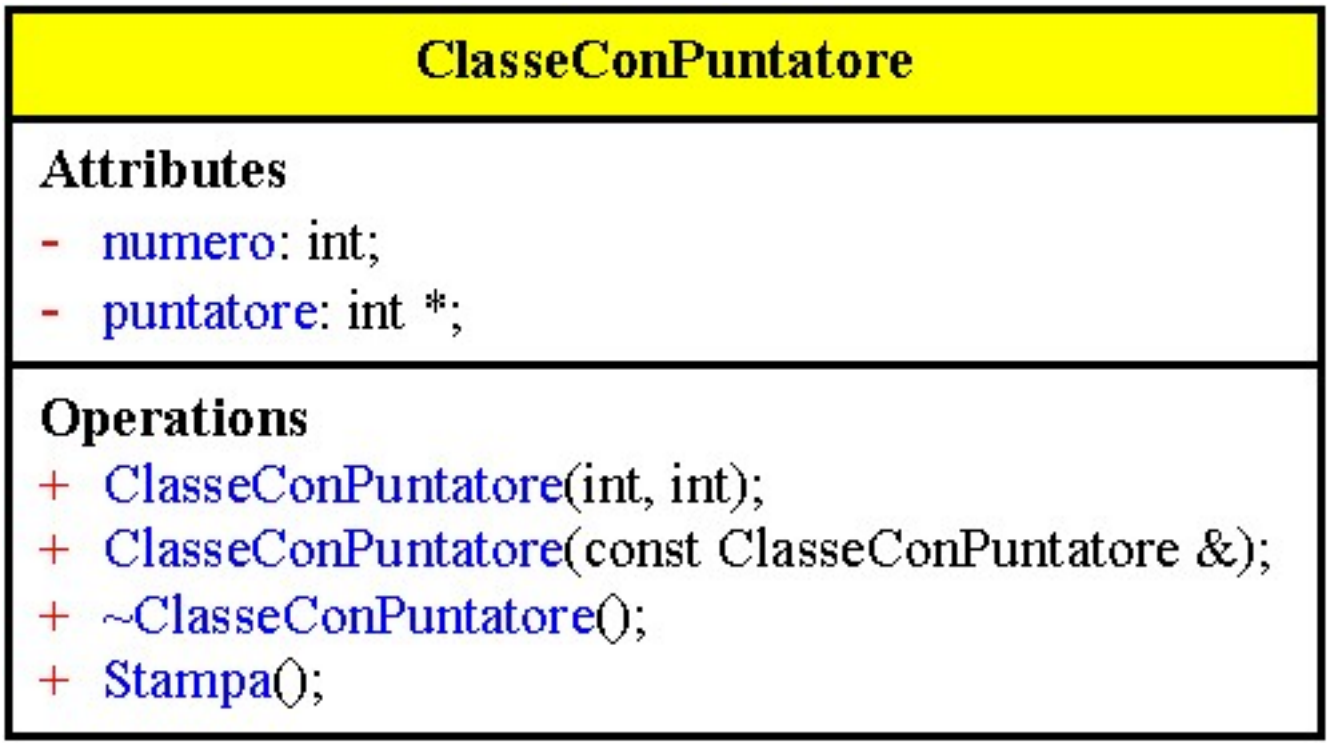
Diagramma U. M. L. delle classi



```
/* Pointcl2.cpp
 * classe con puntatori contenente un costruttore di copia corretto
 * per evitare dangling references (vedere esempio POINTCLS)
 */
#pragma hdrstop
#include <condefs.h>
#include <iostream.h>
#include "ClasseP.h"
//---------------------------------------------------------------------------
USEUNIT("ClasseP.cpp");
//---------------------------------------------------------------------------
#pragma argsused
void Attesa(char *);
int main(int argc, char* argv[]) {
  cout << "Sto creando l'oggetto A" << endl;
  ClasseConPuntatore oggA (5, 8);
  {
    cout << endl;
    cout << "Sto creando l'oggetto B" << endl;
    ClasseConPuntatore oggB = oggA;
    oggB.Stampa();
    cout << endl;
    cout << "L'oggetto B sta per essere distrutto" << endl;
  }
  cout << endl;
  cout << "L'oggetto A e' corretto (niente puntatore dangling)" << endl;
  oggA.Stampa();
  cout << endl;
  cout << "L'oggetto A sta per essere distrutto" << endl;
  Attesa("terminare");
  return 0;
}
void Attesa(char * str) {
  cout << "\n\n\tPremere return per " << str;
  cin.get();
}
```

```
/* ClasseP.h
 */
//---------------------------------------------------------------------------
#ifndef ClassePH
#define ClassePH
class ClasseConPuntatore  {
  int   numero;
  int*  puntatore;
public:
  // costruttore
  ClasseConPuntatore (int, int);
  // costruttore di copia
  ClasseConPuntatore(const ClasseConPuntatore &);
  // distruttore
  ~ClasseConPuntatore();
  void Stampa();
};
//---------------------------------------------------------------------------
#endif


//---------------------------------------------------------------------------
/* ClasseP.cpp
 */
//---------------------------------------------------------------------------
#pragma hdrstop
#include <iostream.h>
#include "ClasseP.h"
//---------------------------------------------------------------------------
#pragma package(smart_init)
ClasseConPuntatore::ClasseConPuntatore (int a, int b) {
  numero = a;
  puntatore = new int;
  *puntatore = b;
  cout << "Nuovo puntatore - indirizzo: " << puntatore
       << ", valore:" << *puntatore << endl;
}
// costruttore di copia
ClasseConPuntatore::ClasseConPuntatore(const ClasseConPuntatore& pc) {
  // copia i membri
  numero = pc.numero;
  // alloca un nuovo puntatore e copia il valore
  puntatore = new int;
  *puntatore = *(pc.puntatore);
  cout << "Nuovo puntatore - indirizzo: " << puntatore
       << ", valore: " << *puntatore << endl;
}
ClasseConPuntatore::~ClasseConPuntatore() {
  cout << "Distruzione puntatore - indirizzo: " << puntatore
       << ", valore:" << *puntatore << endl;
  delete puntatore;
}
void ClasseConPuntatore::Stampa() {
  cout << "Il valore puntato e' " << *puntatore
       << " e il suo indirizzo e' " << puntatore << endl;
}
```

Diagramma U. M. L. delle classi

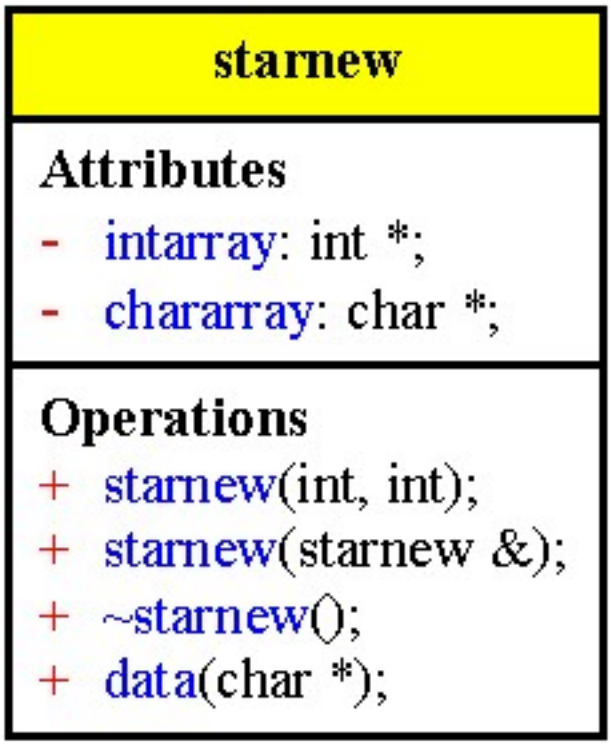


Ritorna

```
/* PStarnew.cpp
 * Perche` return *new solitamente non funziona
 */
#pragma hdrstop
#include <condefs.h>
#include <iostream.h>
#include "Starnew.h"
//---------------------------------------------------------------------------
USEUNIT("Starnew.cpp");
//---------------------------------------------------------------------------
#pragma argsused
starnew return_starnew();
void Attesa(char *);
int main(int argc, char* argv[]) {
  starnew A = return_starnew();
  A.data("A");
  starnew &B = *new starnew(6, 6);
  B.data("B");
  delete &B;
  // Chiamata esplicita al distruttore per il riferimento
  Attesa("terminare");
  return 0;
}
starnew return_starnew() {
  cout << "return_starnew, prima di return *new starnew(5, 5);" << endl;
  return *new starnew(5, 5);
}
void Attesa(char * str) {
  cout << "\n\n\tPremere return per " << str;
  cin.get();
}
/*
 * All'interno della funzione return_starnew(), un oggetto viene creato nella
 * memoria libera, e un puntatore a tale oggetto viene prodotto da new.
 * Quando il puntatore viene derefenziato e ritornato per mezzo della
 * istruzione return *new, una copia della struttura viene ritornata e copiata
 * nella struttura di A. La copia include i puntatori a due array, e quindi
 * viene chiamato il distruttore per A, i due array vengono distrutti.
 * Tuttavia, la struttura originale creata nella memoria libera non viene mai
 * distrutta! Infatti, tale distruzione non e` possibile, poiche` l'indirizzo
 * della struttura viene perso non appena l'istruzione return *new viene
 * eseguita. Quindi, nessun distruttore viene mai chiamato per liberare la
 * memoria allocata per la struttura all'interno della funzione
 * return_starnew().
 */
```

```
/* Starnew.h
 */
//---------------------------------------------------------------------------
#ifndef StarnewH
#define StarnewH
class starnew {
  int *intarray;
  char *chararray;
public:
  void data(char *msg = "");
  starnew(int, int);
  starnew(starnew &);
  ~starnew();
};
//---------------------------------------------------------------------------
#endif


//---------------------------------------------------------------------------
/* Starnew.cpp
 */
//---------------------------------------------------------------------------
#pragma hdrstop
#include <iostream.h>
#include "Starnew.h"
//---------------------------------------------------------------------------
#pragma package(smart_init)
void starnew::data(char *msg) {
  cout << msg << ":\n\tthis = " << (unsigned int) this
       << " intarray = "  << (int) intarray
       << " chararray = " << (int) chararray << endl;
}
starnew::starnew(int intsize, int charsize) {
  intarray  = new int[intsize];
  chararray = new char[charsize];
  data("Costruttore chiamato");
}
starnew::starnew(starnew &rval) {
  rval.data("Argomento del costruttore di copia");
  intarray  = rval.intarray ;
  chararray = rval.chararray;
  data("Dopo il costruttore di copia");
}
starnew::~starnew() {
  delete []intarray;
  delete []chararray;
  data("Distruttore chiamato");
}
```

Diagramma U. M. L. delle classi



Funzioni amiche

Talvolta è necessario che elementi privati di una classe siano accessibili ad altre funzioni; il rendere tali elementi pubblici non è sempre una buona idea in quanto, così facendo, si permette al programmatore di modificare i dati.

Poiché non è possibile che una funzione sia membro di due classi sono state introdotte le funzioni amiche che sono funzioni che non sono membro di una classe ma che sono ugualmente autorizzate ad accedere alla parte privata della classe ospite.

Una funzione amica è caratterizzata dalla parola chiave **friend.**

Esempio:

```
class X;
class Y {
        float ratio;
public:
        friend float h(X x, Y  y);
};
class X {
        int num, denom;
public:
        friend float h(X x, Y y);
};
float h(X x, Y y)
{
        y.ratio = x.num / x.denom;
}
```

Le funzioni amiche hanno gli stessi privilegi di accesso di una funzione membro ma non sono associate ad un oggetto della classe ospite; quindi non è possibile chiamare le funzioni membro della classe ospite senza associare le funzioni agli oggetti.

La classe ospite controlla i privilegi di altre funzioni sui propri dati privati, e quindi conosce sempre chi ha il permesso di modificarli..

Le funzioni amiche non sono membro della classe ma possono essere membro di altre classi; infatti un'intera classe può essere dichiarata amica

Un esempio in cui l'uso di friend risulta utile è quello della suddivisione di un'astrazione in due classi:

- una classe d'interfaccia non soggetta a modifiche
- una classe implementativa le cui modifiche non influenzano il resto del programma

É possibile scegliere come funzione amica una singola funzione membro di un'altra classe; in questo caso l'ordine delle dichiarazioni è importante per il compilatore.

Vantaggi delle funzioni amiche

Una funzione amica può accedere alle componenti private e protected di più classi.

Si risparmia l'overhead delle chiamate delle funzioni di interfaccia.

Una funzione membro, sia la classe T, è invocata associata con un oggetto della classe T. Non può essere invocata con un altro oggetto di un altro tipo, sia X, si dovrebbe convertire un oggetto di classe X in un oggetto di classe T.

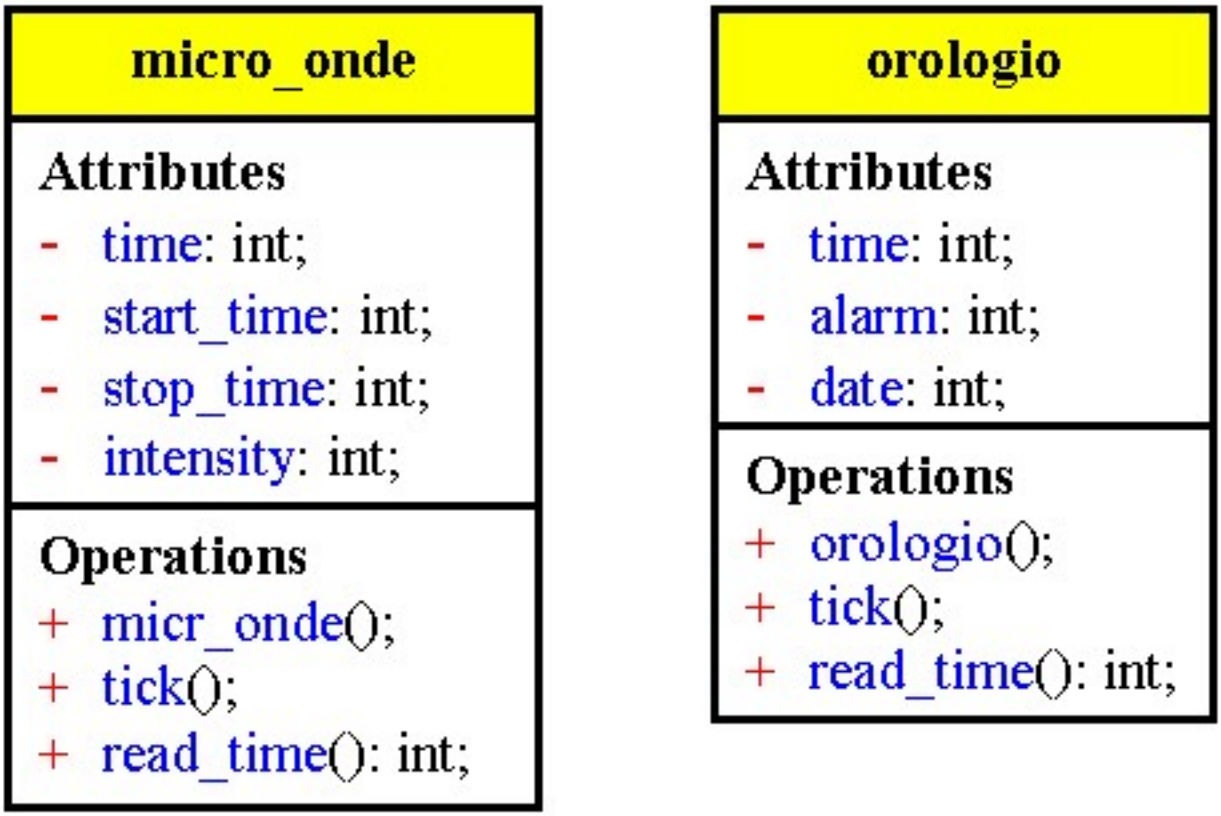
Le funzioni amiche non sono chiamate associate ad un oggetto quindi consentono le conversioni di tipo implicite.

Ritorna

```
/* Friend1.cpp
 * Dimostrazione di funzioni di tipo friend
 * La funzione sincronizza() ha argomenti sia della classe orologio che della
 * classe microonde.
 * La prima volta che sincronizza() è dichiarata come friend in orologio, il
 * compilatore non sa che microonde esiste a meno che non se ne dichiari
 * prima il nome.
 */
#pragma hdrstop
#include <condefs.h>
#include <iostream.h>
#include "Orologio.h"
#include "Microndе.h"
//---------------------------------------------------------------------------
USEUNIT("Orologio.cpp");
USEUNIT("Microndе.cpp");
//---------------------------------------------------------------------------
#pragma argsused
void sincronizza(orologio &, micro_onde &);
void Attesa(char *);
int main(int argc, char* argv[]) {
  orologio A;
  micro_onde B;
  A.tick();
  cout << "Time di A = "  << A.read_time()  << endl;
  A.tick();
  cout << "Time di A = "  << A.read_time()  << endl;
  B.tick();
  cout << "Time di B = "  << B.read_time()  << endl;
  sincronizza(A, B);
  cout << "Time di A = "  << A.read_time()  << endl;
  cout << "Time di B = "  << B.read_time()  << endl;
  Attesa("terminare");
  return 0;
}
void sincronizza(orologio& objA, micro_onde & objB) {
  objA.time = objB.time = 15;   // Definisce uno stato comune
}
void Attesa(char * str) {
  cout << "\n\n\tPremere return per " << str;
  cin.get();
}
/*
 * Le funzioni amiche non sono membro della classe, ma ha gli stessi privilegi
 * di accesso di una funzione membro, ma non e' associata ad un oggetto della
 * classe ospite.
 * Poiche', sincronizza() e' una funzione di tipo friend sia di orologio che di
 * micro_onde, ha accesso agli elementi privati di entrambe.
 * L'operatore & specifica che il paramentro e' passato per riferimento
 * Il riferimento permette di trattare, all'interno della funzione, il nome
 * come se fosse una vera variabile, e non l'indirizzo di una variabile
 * sincronizza() puo' avere accesso e modificare gli elementi privati sia di
 * objA che di objB perche' e' stata dichiarata come funzione amica di entrambe
 * le classi.
 */
```

```
/* Micronde.h
 */
//---------------------------------------------------------------------------
#ifndef MicrondeH
#define MicrondeH
class orologio;
class micro_onde {
public:
  micro_onde();
  void tick();
  int read_time();
  friend void sincronizza(orologio &, micro_onde &);
private:
  int time;
  int start_time;
  int stop_time;
  int intensity;
};
//---------------------------------------------------------------------------
#endif


//---------------------------------------------------------------------------
/* Micronde.cpp
 */
//---------------------------------------------------------------------------
#pragma hdrstop

#include "Micronde.h"
//---------------------------------------------------------------------------
#pragma package(smart_init)
micro_onde::micro_onde() {
  time = 0;
  start_time = stop_time = 0;
  intensity = 0;
}
void micro_onde::tick() {
  time++;
}
int micro_onde::read_time() {
  return time;
}
```

Diagramma U. M. L. delle classi



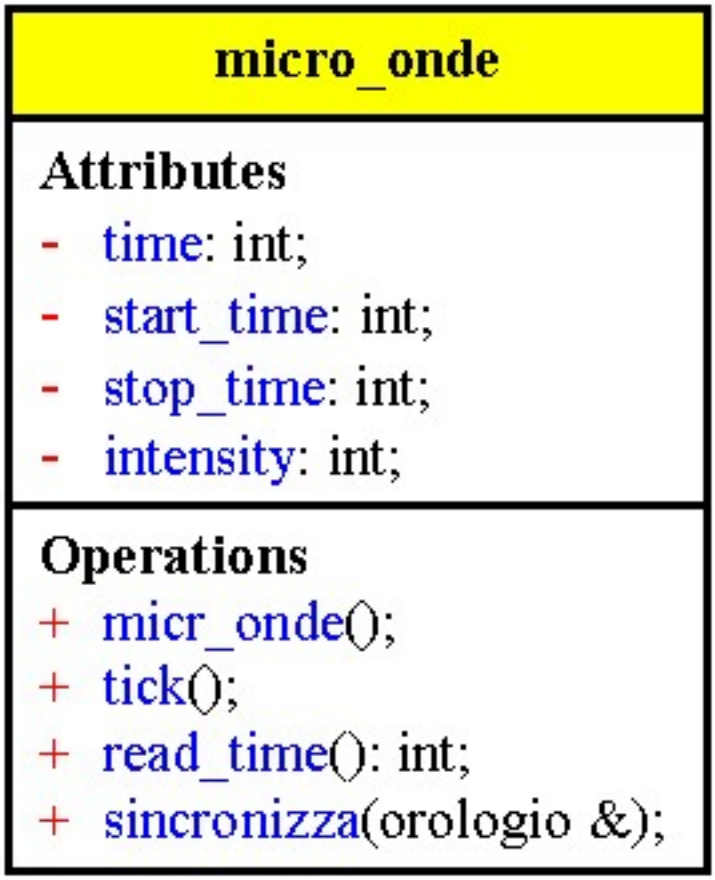
```
/* Orologio.h
 */
//---------------------------------------------------------------------------
#ifndef OrologioH
#define OrologioH
class micro_onde;
class orologio {
public:
  // Il costruttore definisce lo stato iniziale
  orologio();
  void tick();
  int read_time();
  //Dichiara una funzione amica
  friend void sincronizza(orologio&, micro_onde &);
private:
  int time;      // Una misura del tempo
  int alarm;     // Quando l'allarme scatta
  int date;      // Altre cose che orologio dovrebbe conoscere
};
//---------------------------------------------------------------------------
#endif



//---------------------------------------------------------------------------
/* Orologio.cpp
 */
//---------------------------------------------------------------------------
#pragma hdrstop

#include "Orologio.h"
//---------------------------------------------------------------------------
#pragma package(smart_init)
orologio::orologio() {
  time = alarm = date = 0;
}
void orologio::tick() {
  time++;
}
int orologio::read_time() {
  return time;
}
```

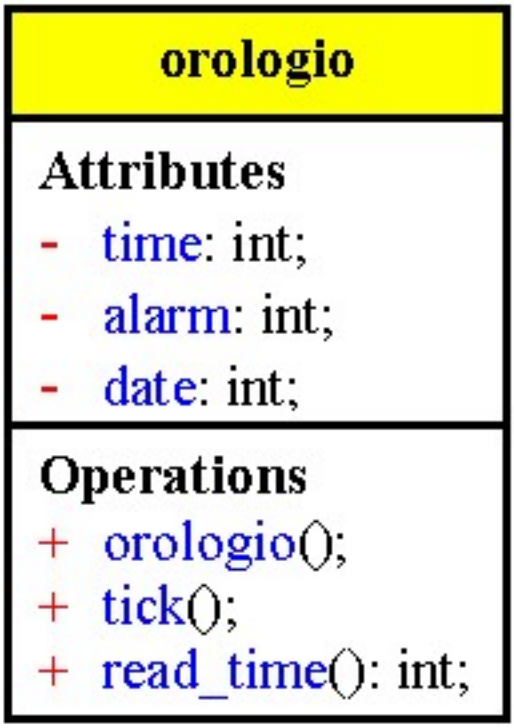
Ritorna

```
/* Friend2.cpp
 * Rendere un intera classe amica
 */
#pragma hdrstop
#include <condefs.h>
#include <iostream.h>
#include "Orologio.h"
#include "Micronde.h"
//---------------------------------------------------------------------------
USEUNIT("Orologio.cpp");
USEUNIT("Micronde.cpp");
//---------------------------------------------------------------------------
#pragma argsused
void Attesa(char *);
int main(int argc, char* argv[]) {
  orologio A;
  micro_onde B;
  A.tick();
  cout << "Time di A = "  << A.read_time()  << endl;
  A.tick();
  cout << "Time di A = "  << A.read_time()  << endl;
  B.tick();
  cout << "Time di B = "  << B.read_time()  << endl;
  B.sincronizza(A);
  cout << "Time di A = "  << A.read_time()  << endl;
  cout << "Time di B = "  << B.read_time()  << endl;
  Attesa("terminare");
  return 0;
}
void Attesa(char * str) {
  cout << "\n\n\tPremere return per " << str;
  cin.get();
}
/*
 * La funzione sincronizza() ha un solo argomento, poiche' una funzione membro
 * conosce gia' l'oggetto per cui e' chiamata
 * Soluzione alternativa:
 * Dichiarare un'intera classe amica e rendere sincronizza() funzione membro
 * di tale classe.
 */
```

```
/* Micronde.h
 */
//---------------------------------------------------------------------------
#ifndef MicrondeH
#define MicrondeH
class orologio;
class micro_onde {
public:
  micro_onde();
  void tick();
  int read_time();
  void sincronizza(orologio &);
private:
  int time;
  int start_time;
  int stop_time;
  int intensity;
};
//---------------------------------------------------------------------------
#endif


//---------------------------------------------------------------------------
/* Micronde.cpp
 */
//---------------------------------------------------------------------------
#pragma hdrstop
#include "Orologio.h"
#include "Micronde.h"
//---------------------------------------------------------------------------
#pragma package(smart_init)
micro_onde::micro_onde() {
  time = 0;
  start_time = stop_time = 0;
  intensity = 0;
}
void micro_onde::tick() {
  time++;
}
int micro_onde::read_time() {
  return time;
}
void micro_onde::sincronizza(orologio & WA) {
  time = WA.time = 15;          // Setta uno stato comune
}
```

Diagramma U. M. L. delle classi





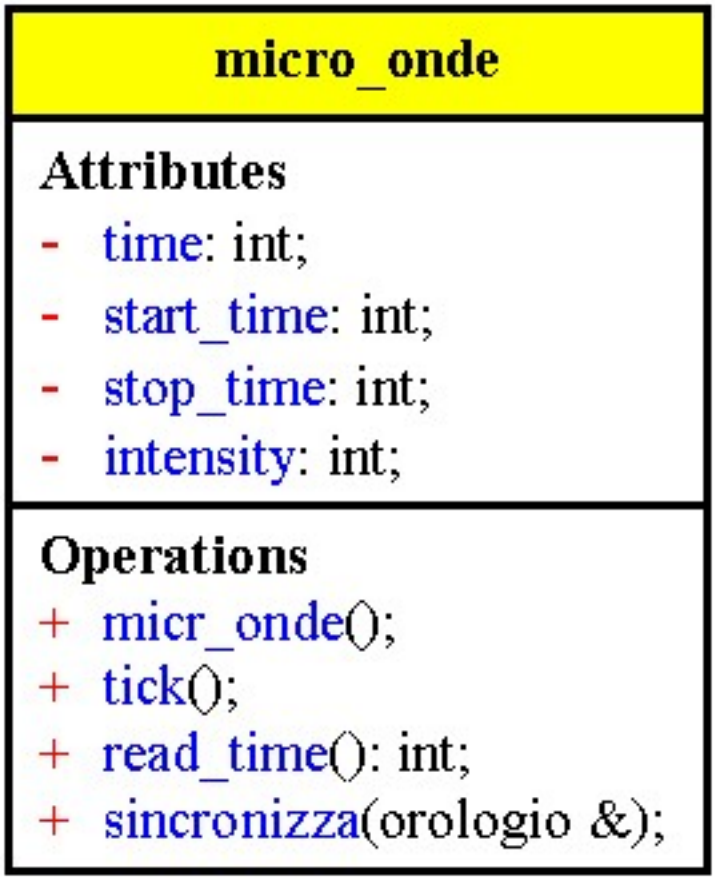
```
/* Orologio.h
 */
//---------------------------------------------------------------------------
#ifndef OrologioH
#define OrologioH
class orologio {
public:
  // Il costruttore definisce lo stato iniziale
  orologio();
  void tick();
  int read_time();
  // Permette a tutti i membri di micro_onde di accedere agli elementi
  // privati di orologio.
  friend class micro_onde;
private:
  int time;      // Una misura del tempo
  int alarm;     // Quando l'allarme scatta
  int date;      // Altre cose che orologio dovrebbe conoscere
};
//---------------------------------------------------------------------------
#endif


//---------------------------------------------------------------------------
/* Orologio.cpp
 */
//---------------------------------------------------------------------------
#pragma hdrstop
#include "Orologio.h"
//---------------------------------------------------------------------------
#pragma package(smart_init)
orologio::orologio() {
  time = alarm = date = 0;
}
void orologio::tick() {
  time++;
}
int orologio::read_time() {
  return time;
}
```

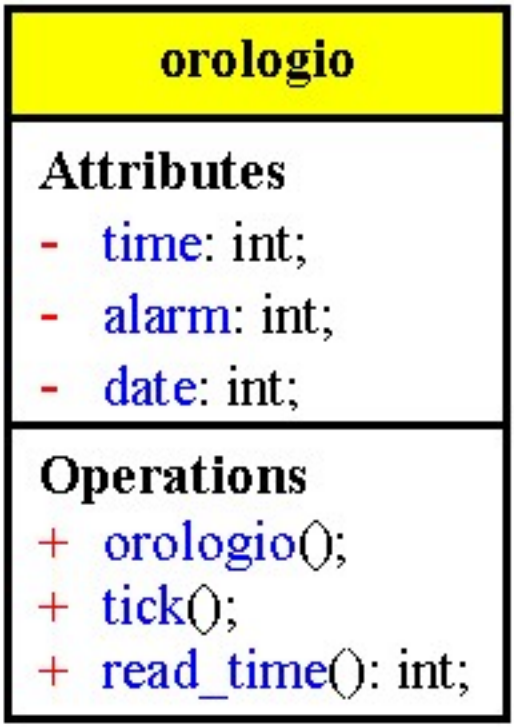
Ritorna

```
/* Friend3.cpp
 * Una funzione membro friend
 */
#pragma hdrstop
#include <condefs.h>
#include <iostream.h>
#include "Orologio.h"
#include "Micronde.h"
//---------------------------------------------------------------------------
USEUNIT("Micronde.cpp");
USEUNIT("Orologio.cpp");
//---------------------------------------------------------------------------
#pragma argsused
void Attesa(char *);
int main(int argc, char* argv[]) {
  orologio A;
  micro_onde B;
  A.tick();
  cout << "Time di A = "  << A.read_time()  << endl;
  A.tick();
  cout << "Time di A = "  << A.read_time()  << endl;
  B.tick();
  cout << "Time di B = "  << B.read_time()  << endl;
  B.sincronizza(A);
  cout << "Time di A = "  << A.read_time()  << endl;
  cout << "Time di B = "  << B.read_time()  << endl;
  Attesa("terminare");
  return 0;
}
void Attesa(char * str) {
  cout << "\n\n\tPremere return per " << str;
  cin.get();
}
/*
 * La funzione membro sincronizza() non puo' essere online poiche' la definizione
 * contiene dei riferimenti a elementi della classe orologio, che non e' ancora
 * dichiarata; la definizione deve apparire dopo la dichiarazione della classe
 * orologio.
 * Se il parametro di sincronizza() fosse stato passato per valore il
 * compilatore non avrebbe saputo a quel punto la dimensione dell'oggetto
 * orologio ed avrebbe riscontrato un errore
 */
```

```
/* Micronde.h
 */
//---------------------------------------------------------------------------
#ifndef MicrondeH
#define MicrondeH
class orologio;         // Dichiarazione del nome della classe

class micro_onde {
public:
  micro_onde();
  void tick();
  int read_time();
  void sincronizza(orologio &);
private:
  int time;
  int start_time;
  int stop_time;
  int intensity;
};
//---------------------------------------------------------------------------
#endif



//---------------------------------------------------------------------------
/* Micronde.cpp
 */
//---------------------------------------------------------------------------
#pragma hdrstop
#include "Orologio.h"
#include "Micronde.h"
//---------------------------------------------------------------------------
#pragma package(smart_init)
micro_onde::micro_onde() {
  time = 0;
  start_time = stop_time = 0;
  intensity = 0;
}
void micro_onde::tick() {
  time++;
}
int micro_onde::read_time() {
  return time;
}
void micro_onde::sincronizza(orologio& WA) {
  time = WA.time = 15;          // Setta uno stato comune
}
```

Diagramma U. M. L. delle classi





```
/* Orologio.h
 */
//---------------------------------------------------------------------------
#ifndef OrologioH
#define OrologioH
#include "Micronde.h"
class orologio {
public:
  // Il costruttore definisce lo stato iniziale
  orologio();
  void tick();
  int read_time();
  // Permette a tutti i membri di micro_onde di accedere agli elementi
  // privati di orologio.
  friend void micro_onde::sincronizza(orologio &);
private:
  int time;      // Una misura del tempo
  int alarm;     // Quando l'allarme scatta
  int date;      // Altre cose che orologio dovrebbe conoscere
};
//---------------------------------------------------------------------------
#endif



//---------------------------------------------------------------------------
/* Orologio.cpp
 */
//---------------------------------------------------------------------------
#pragma hdrstop
#include "Orologio.h"
//---------------------------------------------------------------------------
#pragma package(smart_init)
orologio::orologio() {
  time = alarm = date = 0;
}
void orologio::tick() {
  time++;
}
int orologio::read_time() {
  return time;
}
```

Overloading dei costruttori

Il tipo di overloading più comune riguarda i costruttori; normalmente, in una classe è richiesto più di un costruttore, per gestire diversi tipi di inizializzazioni.
All'interno di una classe l'overloading di una funzione è automatico, è sufficiente definire più funzioni con lo stesso nome.
Esempio:

```
class intero {
        int i;
public:
        intero() { i = 0; }
        intero(int j) { i = j; }
        intero(double d) { i = (int) d; }
};
```

Assegnamento di un oggetto

Il compilatore crea una funzione di default se viene effettuato un assegnamento e operator=() non è stato definito.
L'operatore di default effettua una copia bit a bit per i tipi predefiniti e una copia membro a membro per i tipi definit dal programmatore
operator=() non può essere ereditato in quanto deve svolgere gli stessi compiti del distruttore e del costruttore di copia
operator=() possiede un valore di ritorno, nel caso più comune viene ritornata una copia del nuovo oggetto assegnato per mezzo dell'istruzione return *this; in questo modo è possibile avere assegnamenti in sequenza
L'operator=() è chiamato quando un oggetto già inizializzato viene assegnato a un altro oggetto già inizializzato.

```
X d;
X e = d;     // X(X&) chiamato
X f;
f = d;       // X::operator=(X&) chiamato
```

Se f contiene puntatori oppure oggetti membro, operator=() ha la responsabilità di gestirli
Analisi di un assegnamento:

```
vec func1(vec value);
vec A(4, 3);
vec B;
B = func1(A);
```

All'inizio della funzione viene allocato spazio sullo stack per la struttura dell'oggetto value
Il costruttore di copia viene chiamato per value con A come argomento in modo che A possa essere copiato all'interno della funzione.
L'indirizzo della variabile temporanea viene passata alla funzione
Al termine della funzione viene chiamato il costruttore di copia per la variabile temporanea con value come argomento
Esso copia il valore di ritorno all'esterno della funzione
Il valore di ritorno di func1() viene assegnato a B perciò l'indirizzo della variabile temporanea viene passato a vec::operator=()
operator=() viene chiamato come funzione membro per B
Poiché B può essere legato a nuovi dati, la vecchia rappresentazione di B deve essere prima cancellata

Nuova memoria viene allocata per contenere il nuovo vec e i dati vengono copiati
operator=() ritorna una copia di se stesso chiamando il costruttore di copia per una seconda variabile temporanea, con il proprio indirizzo come argomento.
In questo caso tale variabile temporanea viene ignorata in quanto non è utilizzata, il distruttore la cancellerà quando non sarà più visibile.

Conteggio dei riferimenti

Possono insorgere problemi se si ha una classe con un puntatore come membro, in questo caso il comportamento di default è di copiare i membri, cioè i puntatori e non i valori a cui puntano, così si finisce per avere due oggetti con un membro che si riferisce alla stessa locazione di memoria, con la conseguenza che se uno degli oggetti cambia questo valore entrambi sono coinvolti.
La tecnica di conteggio dei riferimenti indica quanti oggetti stanno utilizzando i dati
Quando un nuovo oggetto viene legato a una particolare struttura il contatore dei riferimenti è incrementato, mentre quando un oggetto viene liberato dal legame con la struttura il contatore di riferimenti viene decrementato.
Il distruttore decrementa semplicemente il contatore di riferimenti e distrugge i dati soltanto quando questo numero diventa zero.
Il costruttore di copia o l'operatore di assegnamento incrementa il contatore di riferimenti dell'argomento e copia il puntatore.

Alias

Definire due oggetti che puntano agli stessi dati non è generalmente una buona pratica di programmazione, poiché ogni modifica apportata a un oggetto si riflette sull'altro.
Metodi per impedire i sinonimi
1 - non permettere assegnamenti del tipo A = B; creando uno schema per distinguere le variabili temporanee dalle "vere" variabili.
2 - creare uno schema che alloca nuova memoria e duplica l'oggetto quando l'utente cerca di scrivere in uno spazio di memoria puntato da due oggetti.

Ritorna

```
/* Mess.cpp
 * Classe messaggio
 */
#pragma hdrstop
#include <condefs.h>
#include <iostream.h>
#include "Messaggio.h"
//---------------------------------------------------------------------------
USEUNIT("Messaggio.cpp");
//---------------------------------------------------------------------------
#pragma argsused
void Attesa(char *);
int main(int argc, char* argv[]) {
  // definisce due oggetti della classe messaggio
  messaggio msg1("Annuncio pubblicitario");
  messaggio msg2 = msg1;
  msg1.SetMess ("Annuncio Vendite");

  // Mostra i messaggi
  cout  << msg1.GetMess() << "    " <<  msg2.GetMess() << endl;
  messaggio msg3("Vuoto");
  msg3 = msg2 = msg1;
  cout << msg3.GetMess() << endl;
  Attesa("terminare");
  return 0;
}
void Attesa(char * str) {
  cout << "\n\n\tPremere return per " << str;
  cin.get();
}
```

```
/* Messaggio.h
 */
//---------------------------------------------------------------------------
#ifndef MessaggioH
#define MessaggioH
class messaggio {
public:
  // solito costruttore con un argomento
  messaggio (char *);
  // costruttore di copia
  messaggio (const messaggio &);
  // Provare a ritornare per valore
  messaggio &operator=(messaggio &);
  // distruttore
  ~messaggio ();
  char * GetMess ();
  void  SetMess (char *);
private:
  char  * msg;
};
//---------------------------------------------------------------------------
#endif



//---------------------------------------------------------------------------
/* Messaggio.cpp
 */
//---------------------------------------------------------------------------
#pragma hdrstop

#include "Messaggio.h"
//---------------------------------------------------------------------------
#pragma package(smart_init)
#include        <iostream.h>
#include        <string.h>
messaggio::messaggio (char * n) {
  msg = new char [20];
  strcpy (msg, n);
  cout << "*** Chiamata del costruttore normale ***" << endl;
}
messaggio::messaggio (const messaggio &b) {
  msg = new char [20];
  strcpy (msg, b.msg);
  cout << "*** Chiamata del costruttore di copia ***" << endl;
}
messaggio & messaggio::operator=(messaggio &b) {
  strcpy(msg, b.msg);
  cout << "*** Chiamata operatore di assegnamento ***" << endl;
  return *this;
}
messaggio::~messaggio () {
   delete []msg;
   cout << "*** Chiamata del distruttore ***" << endl;
}
char * messaggio::GetMess () {
  return msg;
}
void messaggio::SetMess (char * n) {
  strcpy (msg, n);
}
```

Diagramma U. M. L. delle classi

Overloading degli operatori

Quando si applica l'overloading a un operatore, si crea una funzione, che è esattamente uguale a qualunque altra funzione, tranne per il nome.
Il nome della funzione è sempre composto dalla parola chiave operator, seguita dai caratteri usati per l'operatore.
Per eseguire l'overloading di un operatore si definisce una funzione che il compilatore deve chiamare quando l'operatore è usato con i tipi di dati appropriati.
Gli unici operatori che non si possono sovraccaricare sono:

. .* :: ?: sizeof

Sintassi per una funzione amica:
Tipo_di_ritorno operator@(lista_argomenti) { corpo_della_funzione }
Una funzione amica ha un argomento per un operatore unario e due argomenti per un operatore binario.
Sintassi per un membro della classe:
tipo_di_ritorno nome_classe::operator@(argomento) { corpo_della_funzione }
Una funzione membro non ha argomenti per un operatore unario e ha un argomento per un operatore binario; infatti uno degli argomenti è implicitamente l'oggetto per cui è stata chiamata.
Nota: É bene cercare di sovraccaricare gli operatori con altri che abbiano una semantica il più possibile simile anche se questo non è obbligatorio.
Poiché gli operatori overloaded possono essere usati in espressioni complesse e non necessariamente da soli, è importante che essi ritornino il valore risultante di una operazione.
Scegliere funzioni amiche o membro per l'overloading degli operatori
Se si usa una funzione membro e un argomento di diverso tipo, la funzione membro permette al nuovo tipo di trovarsi solo alla destra dell'operatore A + 2 è legale mentre 2 + A non lo è
Per una funzione amica entrambe le espressioni sono legali
Se la sintassi richiede che gli argomenti siano indipendenti si utilizzi una funzione amica

Perché i riferimenti sono essenziali

Il compilatore chiama, in maniera trasparente, la funzione overloaded, quando riconosce i tipi di dati appropriati per l'operatore.
Esempio:

```
class plus {
        int i;
public:
        plus operator+(plus a) {
                i += a.i;
                return i;
        }
};
```

La funzione operator+() ha un solo argomento, che è il valore alla destra del segno +.
La funzione è chiamata per l'oggetto alla sinistra del segno +.
La funzione operator+() crea un nuovo oggetto plus, chiamando il costruttore plus il cui argomento è la somma dei dati privati dei due oggetti.
Il costruttore plus crea un nuovo oggetto temporaneo plus che viene copiato all'esterno della funzione, come il valore di ritorno dell'operatore.Tutto funziona perché gli oggetti sono passati e ritornati per valore.

Vi sono molte situazioni in cui occorre passare l'indirizzo di un oggetto
La sintassi A + &B funziona genera confusione
I riferimenti permettono di passare indirizzi agli operatori overloaded e di mantenere la notazione A+B

Personalizzazione delle funzioni iostream

Per aggiungere il supporto dell'input e dell'output standard per i tipi di dati definiti dall'utente è necessario definire alcuni nuovi operatori globali, non membri di classi.
Questi operatori hanno due parametri, lo stream e il dato e ritornano lo stream per permettere il concatenamento degli operatori.
L'oggetto stream è passato e ritornato per indirizzo evitando copie locali.
Esempio:

```
ostream &operator<<(ostream &s, Data d) {
        s << d.giorno() << '/'<< d.mese() << '/'d.anno();
        return s;
}
istream &operator >> (istream &s, Data &d) {
        int g, m, a;
        s >> g >> m >> a;
        // Esegue i controlli
        d.SetData(g, m, a);
        return s;
}
```

L'operatore operator<<() è già un operatore oveloaded per la classe iostream, ma si può avere un nuovo overloading per la propria classe.
Gli operatori overloaded operator<<() e operator>>() devono essere funzioni amiche globali, e devono avere come argomento un oggetto iostream seguito da un oggetto di un tipo definito dall'utente; la funzione deve ritornare lo stesso oggetto stream che ha come argomento. In effetti, ciascun argomento è aggiunto allo stream, e poi lo stream è passato all'elemento successivo della linea.

operator ++

Il compilatore distingue la forma prefissa dalla postfissa di questi operatori chiamando la funzione rispettivamente senza argomenti o con un argomento intero
L'argomento intero non ha un identificatore e non è usato nella funzione, serve solo per distinguere i due tipi di funzione

operator[]

L'espressione A[i][j] = 1; viene valutata dal compilatore:
- il compilatore trova l'oggetto A
- scandisce l'insieme di parentesi quadre associate all'oggetto
- chiama A.operator[](i)
- il risultato dell'operator[] interno deve produrre un oggetto o un riferimento ad un oggetto e anche nella classe di tale oggetto deve esserci un operator[]
- L'espressione viene dunque valutata in A.operator[](i).operator[](j)

operator()

L'overloading di una chiamata di funzione è prezioso poiché permette di usare la stessa sintassi di un operatore con argomenti multipli

operator virgola

In una sequenza di espressioni separate da una virgola le espressioni sono valutate da sinistra verso destra. Tutti i valori risultanti dalla valutazione delle espressioni sono scartati tranne il valore finale che è il risultato dell'intera sequenza.
Tutte le espressioni, tranne l'ultima sono usate solo per i loro effetti collaterali
Quando l'espressione alla sinistra della virgola è un oggetto, o produce un oggetto, l'operatore virgola può essere overloaded per generare un effetto collaterale.

Puntatori intelligenti

Un puntatore intelligente è un puntatore che è usato su un oggetto della classe e non per puntare ad un oggetto
Esempio: Se C è una classe con operator->() come funzione membro, ed è un membro di tale classe o di qualche altra classe, struttura o unione

```
C cobj;
cobj->el;
è valutato come (cobj.operator->())->el;
```

operator->() deve ritornare uno tra questi due elementi

1. un puntatore a un oggetto della classe contenente l'elemento el
2. un oggetto di un'altra classe, la quale deve contenere una definizione per operator->(). In questo caso operator->() viene chiamato di nuovo per il nuovo oggetto, il processo è ripetuto ricorsivamente finché il risultato è un puntatore a un oggetto della classe contenente l'elemento el

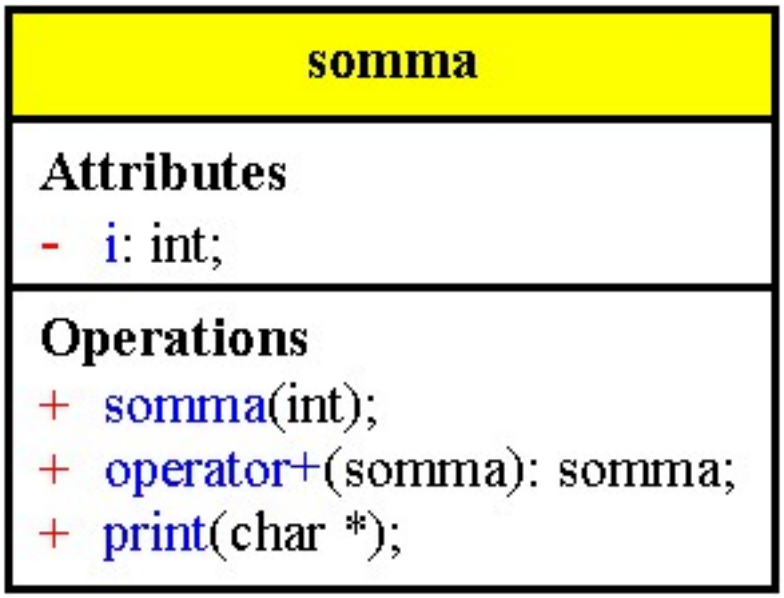
Overloading di new e delete

Quando si ridefiniscono new e delete come membri di una classe, il compilatore chiama queste funzioni solo quando si crea un oggetto di quella classe sullo heap.
Per specificare le versioni globali si deve usare l'operatore :: prima del nome delle funzioni altrimenti vengono usate le versioni locali (e si ottiene una funzione ricorsiva che non termina mai).
Le routine di garbage_collection liberano lo spazio occupato da variabili non utilizzate

Ritorna

```
/* Operpl1.cpp
 * Classe con un operatore + overloaded
 */
#pragma hdrstop
#include <condefs.h>
#include <iostream.h>

//---------------------------------------------------------------------------
#pragma argsused
class somma {
public:
  somma(int x = 0) {i = x; }
  somma operator + (somma arg) {
    return somma(i + arg.i);
  }
  void print(char *msg = "") {
    cout << msg << "  : i = " << i << endl;
  }
private:
  int i;
};
void Attesa(char *);
int main(int argc, char* argv[]) {
  somma A(13), B(34);
  somma C = A + B;
  C.print("C = A + B");
  Attesa("terminare");
  return 0;
}
void Attesa(char * str) {
  cout << "\n\n\tPremere return per " << str;
  cin.get();
}
/*
 * Quando si applica l'overloading a un operatore, si crea una funzione, che
 * e' esattamente uguale a qualunque altra funzione, tranne per il nome.
 * Il nome della funzione e' sempre composto dalla parola chiave operator,
 * seguita dai caratteri usati per l'operatore.
 * Il compilatore chiama, in maniera trasparente, la funzione overloaded,
 * quando riconosce i tipi di dati appropriati per l'operatore.
 * La funzione operator+() crea un nuovo oggetto somma, chiamando il costruttore
 * somma, il cui argomento e' la somma dei dati privati parti dei due oggetti.
 * Il costruttore somma crea un nuovo (temporaneo) oggetto somma, che viene
 * copiato all'esterno della funzione, come il valore di ritorno dell'operatore.
 */
```

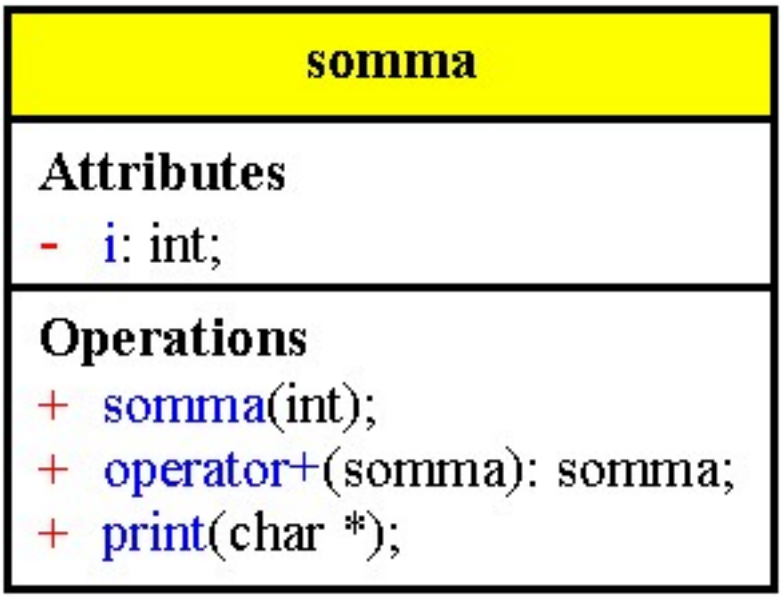
Diagramma U. M. L. delle classi



Ritorna

```
/* Operpl2.cpp
 * Tentativo di usare i puntatori per l'overloading di un operatore
 */
#pragma hdrstop
#include <condefs.h>
#include <iostream.h>
//---------------------------------------------------------------------------
#pragma argsused
class somma {
public:
  somma(int x = 0) { i = x; }
  somma operator +(somma *arg) { return somma(i + arg->i); }
  void print(char *msg = "") {
    cout << msg << "  : i = " << i << endl;
  }
private:
  int i;
};
void Attesa(char *);
int main(int argc, char* argv[]) {
  somma A(13), B(34);
  somma C = A + &B;
  C.print("C = A + B");
  Attesa("terminare");
  return 0;
}
void Attesa(char * str) {
  cout << "\n\n\tPremere return per " << str;
  cin.get();
}
/*
 * Questo programma funziona, ma la sintassi A + &B genera confusione
 */
```

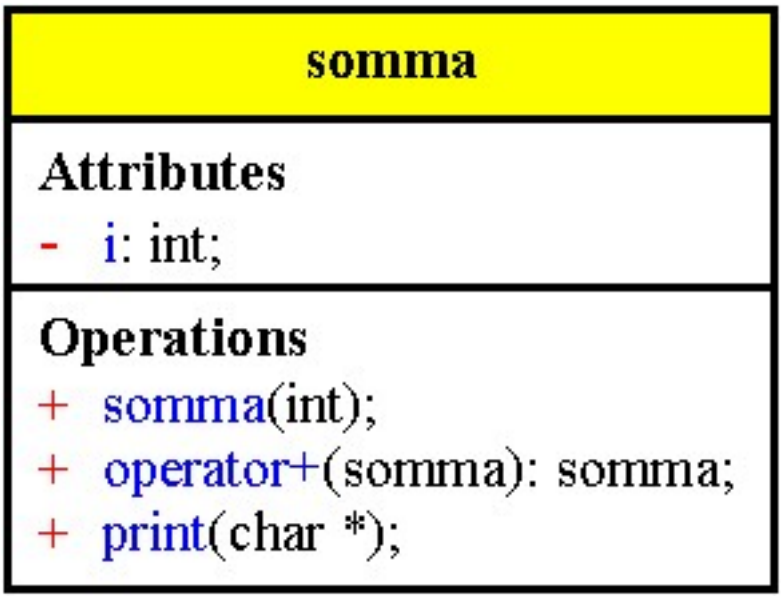
Diagramma U. M. L. delle classi



Ritorna

```
/* Operpl3.cpp
 * Uso dei riferimenti per l'oveloading degli operatori
 * quando occorre passare un indirizzo invece di un valore
 */
#pragma hdrstop
#include <condefs.h>
#include <iostream.h>

//---------------------------------------------------------------------------
#pragma argsused
class somma {
public:
  somma(int x = 0) { i = x; }
  somma operator +(somma &arg) { return somma(i + arg.i); }
  void print(char *msg = "") {
    cout << msg << "  : i = " << i << endl;
  }
private:
  int i;
};
void Attesa(char *);
int main(int argc, char* argv[]) {
  somma A(13), B(34);
  somma C = A + B;       // La sintassi e' chiara come nel passaggio per valore
  C.print("C = A + B");
  Attesa("terminare");
  return 0;
}
void Attesa(char * str) {
  cout << "\n\n\tPremere return per " << str;
  cin.get();
}
/*
 * Con i riferimenti, si puo' usare l'overloading degli operatori e passare
 * indirizzi quando si usano oggetti di grandi dimensioni
 */
```

Diagramma U. M. L. delle classi

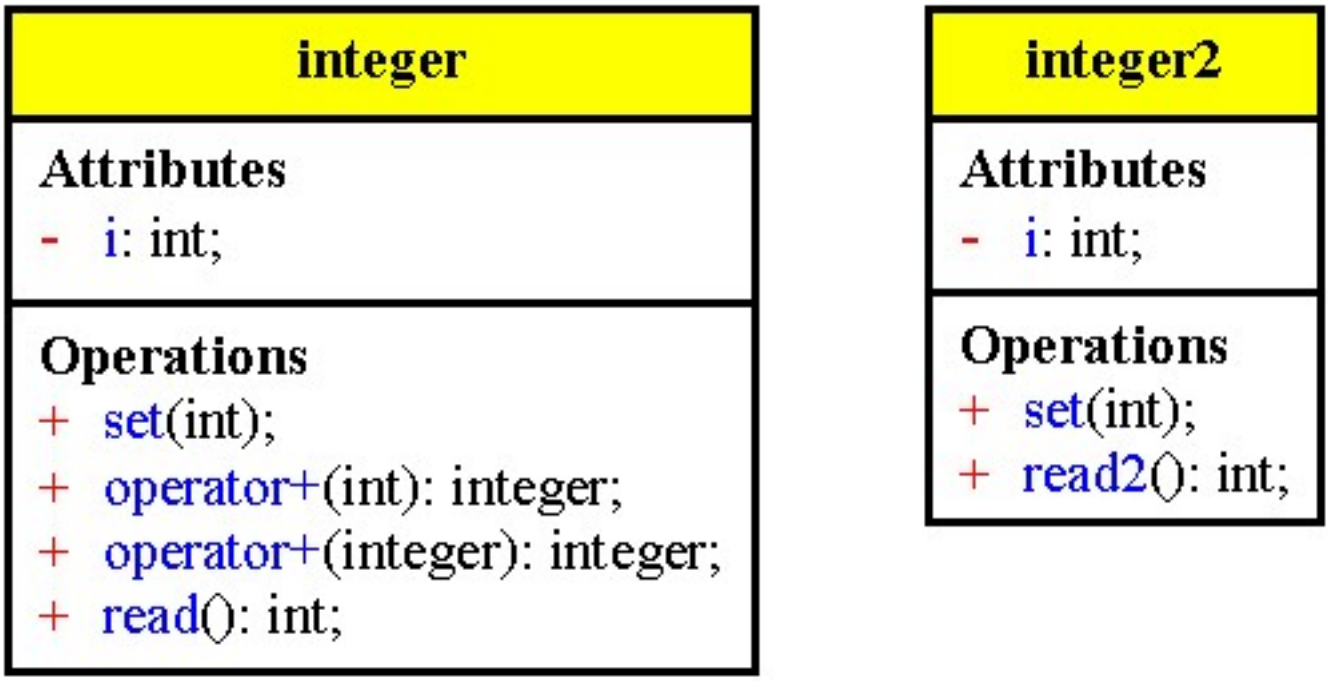


Ritorna

```
/* Friend.cpp
 * Perche' le funzioni amiche sono necessarie
 */
#pragma hdrstop
#include <condefs.h>
#include <iostream.h>
#include "Integer.h"
//---------------------------------------------------------------------------
USEUNIT("Integer.cpp");
//---------------------------------------------------------------------------
#pragma argsused
void Attesa(char *);
int main(int argc, char* argv[]) {
  integer A;
  A.set(10);
  integer B;
  integer C;
  C.set(20);
  B = A + 4;                                  // Legale per la classe integer
  cout << "B = A + 4  =  " << B.read() << endl;
  B = A + C;                                  // Legale per la classe integer
  cout << "B = A + C  =  " << B.read() << endl;
  // B = 4 + A;         // Non legale per la classe integer
  integer2 D;
  D.set(100);
  integer2 E;
  integer2 F;
  F.set(200);
  E = D + 40;                                 // Legale per la classe integer2
  cout << "E = D + 40 =  " << E.read2() << endl;
  E = D + F;                                  // Legale per la classe integer2
  cout << "E = D + F  =  " << E.read2() << endl;
  E = 40 + D;                                 // Legale per la classe integer2
  cout << "E = 40 + D =  " << E.read2() << endl;
  Attesa("terminare");
  return 0;
}
void Attesa(char * str) {
  cout << "\n\n\tPremere return per " << str;
  cin.get();
}
/*
 * Nell'overloading di un operatore alla funzione amica devono essere passati
 * entrambi gli oggetti, mentre la funzione membro necessita di un solo
 * argomento.
 * Se si usa una funzione membro e un argomento di diverso tipo, la funzione
 * membro permette al nuovo tipo di trovarsi solo alla destra dell'operatore.
 * Una funzione amica consente di avere una sintassi piu' naturale
 */
```

```
/* Integer.h
 */
//---------------------------------------------------------------------------
#ifndef IntegerH
#define IntegerH
class integer {
public:
  // Non vi e' un costruttore con un unico argomento di tipo int, e quindi non
  // vi puo' essere una conversione implicita di tipo da int a integer.
  void set(int ii = 0) { i = ii; }
  // Overloading degli operatori con funzioni membro
  integer operator+(int);
  integer operator+(integer);
  int read() { return i; }
private:
  int i;
};
class integer2 {
public:
  void set(int ii = 0) { i = ii; }
  // Overloading degli operatori con funzioni amiche; occorre una funzione per
  // ciascuna possibile combinazione
  friend integer2 operator+(integer2, integer2);
  friend integer2 operator+(integer2, int);
  friend integer2 operator+(int, integer2);
  int read2() { return i; }
private:
  int i;
};
//---------------------------------------------------------------------------
#endif


//---------------------------------------------------------------------------
/* Integer.cpp
 */
//---------------------------------------------------------------------------
#pragma hdrstop
#include "Integer.h"
//---------------------------------------------------------------------------
#pragma package(smart_init)
integer integer::operator+(int x) {
  // Diventa noioso senza un costruttore
  integer result;
  result.set(i + x);
  return result;
}
integer integer::operator+(integer x) {
  integer result;
  result.set( i + x.i );
  return result;
}
integer2 operator+(integer2 x, integer2 y) {
  integer2 result;
  result.set(x.i + y.i);
  return result;
}
integer2 operator+(integer2 x, int a) {
  integer2 result;
  result.set(x.i + a);
  return result;
}
integer2 operator+(int a, integer2 x) {
  integer2 result;
  result.set(x.i + a);
  return result;
}
```

Diagramma U. M. L. delle classi

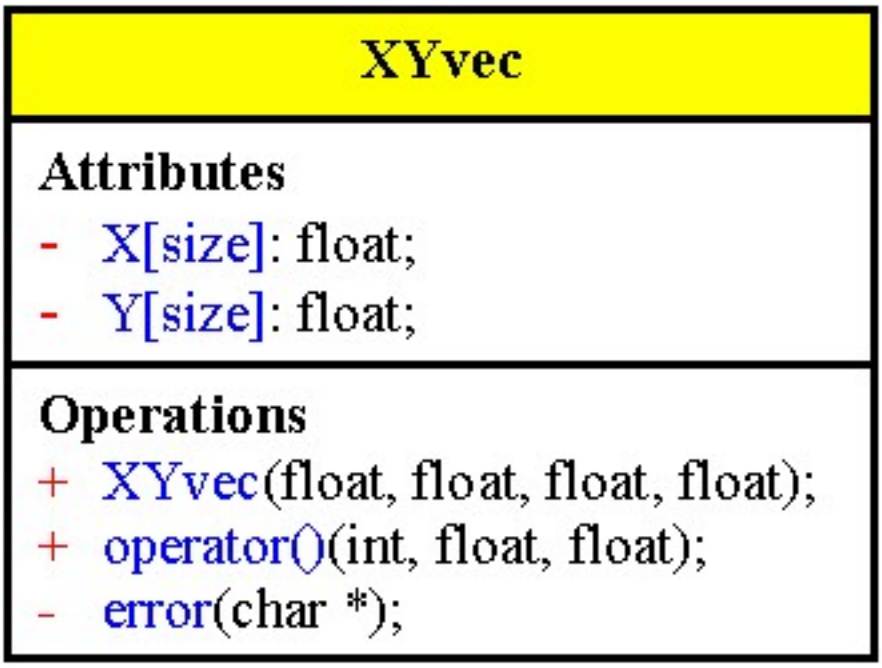


Ritorna

```
/* Xytest.cpp
 * Programma di test per la classe XYvec
 */
#pragma hdrstop
#include <condefs.h>
#include <iostream.h>
#include "Xyvec.h"
//---------------------------------------------------------------------------
USEUNIT("Xyvec.cpp");
//---------------------------------------------------------------------------
#pragma argsused
void Attesa(char *);
int main(int argc, char* argv[]) {
  XYvec A(3.14, 0.47, 2.59);
  A(2, 77.77, 111.9);              // Cambia la coppia #3
  cout << "A = " << A << endl;     // e la stampa
  XYvec B;
  cout << "Inserire " << size*2 << " numeri" << endl;
  cin  >> B;
  cout << "B = " << B << endl;
  Attesa("terminare");
  return 0;
}
void Attesa(char * str) {
  cout << "\n\n\tPremere return per " << str;
  cin.ignore(4, '\n');
  cin.get();
}
```

```
/* Xyvec.h
 * Vettore di coppie X-Y, con funzioni di iostream
 */
#ifndef XyvecH
#define XyvecH
#include <iostream.h>
const size = 5;
class XYvec {
public:
  XYvec(float xinit = 0, float xstep = 0, float yinit = 0, float ystep = 0);
  // Modifica i valori per mezzo dell'overloading dell'operatore di chiamata
  // di funzione
  void operator()(int index, float xval, float yval);
  // Funzioni di iostream
  friend ostream & operator<<(ostream & s, XYvec & v);
  friend istream & operator>>(istream & s, XYvec & v);
private:
  float X[size];
  float Y[size];
  void error(char *msg);
};
#endif
//---------------------------------------------------------------------------
/* Xyvec.cpp
 * Operazioni per il vettore di coppie X-Y
 */
#pragma hdrstop
#include "Xyvec.h"
#pragma package(smart_init)
void Attesa(char *);
void XYvec::error(char * msg) {
  cerr << "Errore in XYvec : " << msg << endl;
  Attesa("terminare");
  exit(1);
}
XYvec::XYvec(float xinit, float xstep, float yinit, float ystep) {
  for(int i = 0; i < size; i++) {
    X[i] = xinit + i * xstep;
    Y[i] = yinit + i * ystep;
  }
}
void XYvec::operator()(int index, float xval, float yval) {
  if(index < 0 || index >= size)
    error("Indice fuori limiti");
  X[index] = xval;
  Y[index] = yval;
}
ostream & operator<<(ostream & s, XYvec & v) {
  s << "\t  X\t\t  Y" << endl;
  s.precision(6);
  for(int i = 0; i < size; i++)
    s << "\t" << v.X[i] << "\t\t" << v.Y[i] << endl;
  return s;
}
istream & operator>>(istream & s, XYvec & v) {
  float val;
  int index = 0;
  while(!s.bad() && !s.eof()) {
    s >> val;
    v.X[index] = val;
    if(s.bad() || s.eof())
       break;
    s >> val;
    v.Y[index++] = val;
    if(index == size)
      break;
  }
  return s;
}
/* operator() e' utile, in quanto e' l'unico operatore che puo' avere un numero
 * arbitrario di argomenti
 * In questo caso e' overloaded per permettere di definire il valore di una
 * particolare coppia X-Y
 */
```

Diagramma U. M. L. delle classi



Ritorna

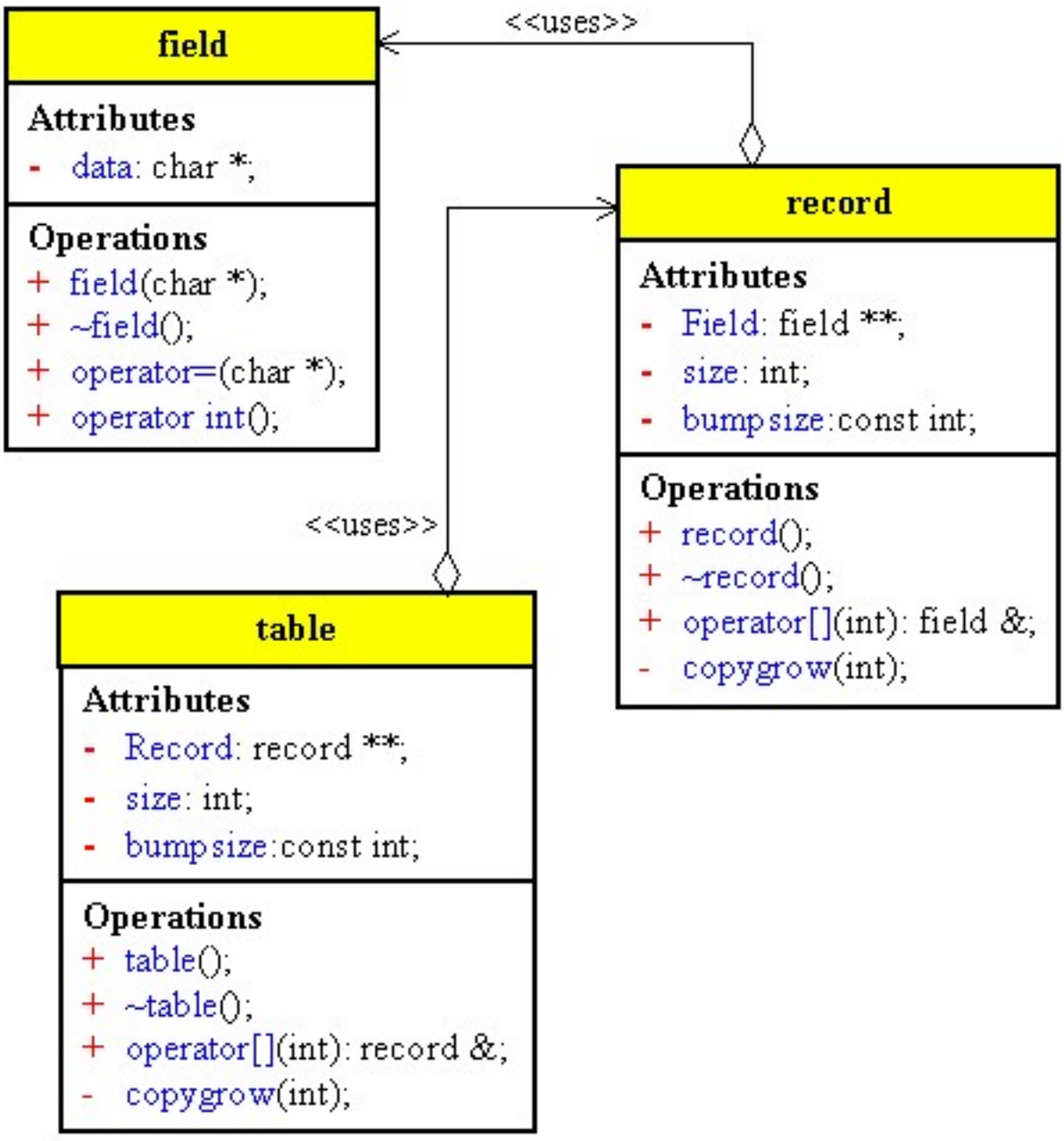
```
/* Brackets.cpp
 * Overloadin di livelli multipli di parentesi quadre
 */
#pragma hdrstop
#include <condefs.h>
#include <iostream.h>
#include <fstream.h>
#include "Table.h"
USEUNIT("Field.cpp");
USEUNIT("Record.cpp");
USEUNIT("Table.cpp");
#pragma argsused
void Attesa(char *);
int main(int argc, char* argv[]) {
  if(argc < 3) {
    cerr << "Uso: brackets file indice";
    Attesa("terminare");     exit(0);
  }
  ifstream in(argv[1]);
  if(!in) {
    cerr << "Errore apertura file" << argv[1];
    Attesa("Terminare");     exit(0);
  }
  const int index = atoi(argv[2]);
  const int bsz = 120;
  char buf[bsz];
  table Tbl;
  int lines = 0;
  while(in.getline(buf, bsz)) {
    int j = 0;
    char *p = strtok(buf, " \t\n");
    while(p) {
      Tbl[lines][j++] = p;
      p = strtok(0, " \t\n");
    }
    lines++;
  }
  int k;      // Stampa l'intera tabella
  for(k = 0; k < lines; k++) {
    for(int i = 0; Tbl[k][i]; i++)
      cout << Tbl[k][i] << " ";
    cout << endl;
  }
  for(k = 0; k < lines; k++)     // Stampa l'elemento indicizzato
    cout << Tbl[k][index] << endl;
  Attesa("terminare");
  return 0;
}
void Attesa(char * str) {
  cout << "\n\n\tPremere return per " << str;
  cin.get();
}
/* Nella classe field viene usata la funzione strdup (ANSI C) che alloca la
 * memoria ed esegue la copia, siccome usa malloc(), all'interno del distruttore
 * deve essere chiamata la funzione free().
 * Viene eseguito l'overloadin di operator=() per poter effettuare l'assegnamento
 * di stringhe di caratteri.
 * Viene creato l'operatore di conversione automatica di tipo a (int) per
 * controllare che field abbia la lunghezza diversa da zero
 * L'operatore overloaded operator<<() e' in realta', una funzione non membro
 * poiche' e' preceduto dalla parola chiave friend; e' questa la forma da usare
 * quando si crea un operatore iostream overloaded per una specifica classe.
 * Gli argomenti indicano che un oggetto ostream si trova a sinistra di <<,
 * e un oggetto field si trova alla sua destra.
 * le classi record e table sono quasi identiche, la caratteristica di entrambe
 * e' che si espandono per adattarsi a qualunque indice si scelga tramite
 * operator[], i limiti si espandono in funzione delle richieste dell'utente.
 * La funzione copygrow() espande l'array in ciascuna classe, l'array
 * inizialmente non ha elementi, e si espande ogni volta che viene scelto un
 * indice superiore alla dimensione corrente.
 * Gli oggetti vengono incrementati della quantit… bumpsize, la quale puo'
 * essere modificata per aumentare l'efficienza a tempo di esecuzione.
 * Per inserire un elemento nella tabellasi usa l'operatore field operator= */
```

```
/*Field.h
 */
//---------------------------------------------------------------------------
#ifndef FieldH
#define FieldH
class field {
public:
  field(char *d = "");
  ~field();
  void operator=(char *);
  operator int();
  friend ostream & operator<<(ostream &, field &);
private:
  char *data;
};
//---------------------------------------------------------------------------
#endif


//---------------------------------------------------------------------------
/* Field.cpp
 */
//---------------------------------------------------------------------------
#pragma hdrstop
#include <iostream.h>
#include "Field.h"
//---------------------------------------------------------------------------
#pragma package(smart_init)
field::field(char *d) : data(strdup(d)) {}
field::~field() {
  free(data);
}
void field::operator=(char *d) {
  free(data);
  data = strdup(d);
}
field::operator int() {
  return strlen(data);
}
ostream & operator<<(ostream & os, field & f) {
  return os << f.data;
}
```

```
/* Record.h
 */
//---------------------------------------------------------------------------
#ifndef RecordH
#define RecordH
#include "Field.h"
class record {
public:
  record() : size(0), Field(0) {}
  ~record();
  field & operator[](int);
private:
  field **Field;
  int size;
  void copygrow(int);
  enum { bumpsize = 10 };               // Quantita' da aumentare
};
//---------------------------------------------------------------------------
#endif


//---------------------------------------------------------------------------
/* Record.cpp
 */
//---------------------------------------------------------------------------
#pragma hdrstop
#include <iostream.h>
#include "Record.h"
//---------------------------------------------------------------------------
#pragma package(smart_init)
void Attesa(char *);
field & record::operator[](int i) {
  if(i < 0) {
    cerr << "Record: indice negativo";
    Attesa("terminare");
    exit(0);
  }
  if(i >= size)
    copygrow(i + bumpsize);
  return *Field[i];
}
void record::copygrow(int newsize) {
  field **temp = new field *[newsize];
  int i;
  for(i = 0; i < size; i++)
    temp[i] = Field[i];
  while(i < newsize)
    temp[i++] = new field;
  size = newsize;
  delete Field;
  Field = temp;
}

record::~record() {
  for(int i = 0; i < size; i++)
    delete Field[i];
  delete Field;
}
```

```
/* Table.h
 */
//---------------------------------------------------------------------------
#ifndef TableH
#define TableH
#include "Record.h"
class table {
public:
  table() : size(0), Record(0) {}
  ~table();
  record & operator[](int);
private:
  record **Record;
  int size;
  void copygrow(int);
  enum { bumpsize = 10 };                        // Quantita' da aumentare
};
//---------------------------------------------------------------------------
#endif


//---------------------------------------------------------------------------
/* Table.cpp
 */
//---------------------------------------------------------------------------
#pragma hdrstop
#include <iostream.h>
#include "Table.h"
//---------------------------------------------------------------------------
#pragma package(smart_init)
void Attesa(char *);
record & table::operator[](int i) {
  if(i < 0) {
    cerr << "Table: indice negativo";
    Attesa("terminare");
    exit(0);
  }
  if(i >= size)
    copygrow(i + bumpsize);
  return *Record[i];
}
void table::copygrow(int newsize) {
  record **temp = new record*[newsize];
  int i;
  for(i = 0; i < size; i++)
    temp[i] = Record[i];
  while( i < newsize)
    temp[i++] = new record;
  size = newsize;
  delete Record;
  Record = temp;
}

table::~table() {
  for(int i = 0; i < size; i++)
    delete Record[i];
  delete Record;
}
```

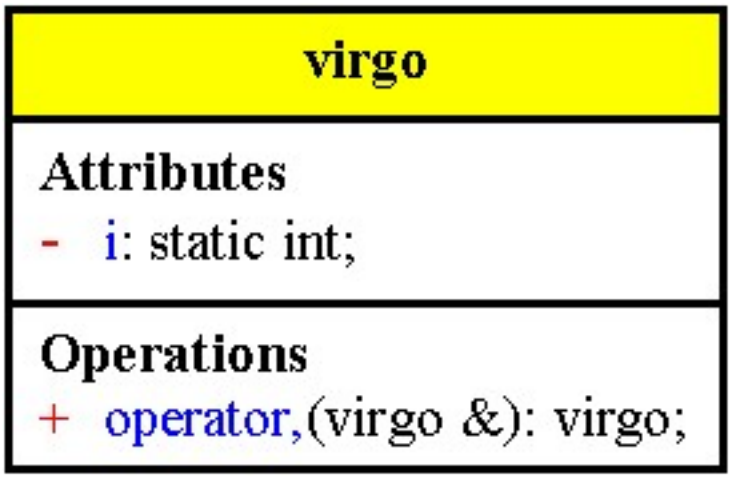
Diagramma U. M. L. delle classi



Ritorna

```
/* Virgo.cpp
 * Overloading dell'operatore virgola
 */
#pragma hdrstop
#include <condefs.h>
#include <iostream.h>
//---------------------------------------------------------------------------
#pragma argsused
class virgo {
  static int i;
public:
  virgo operator,(virgo &);
};
int virgo::i = 0;                       // Definizione di un membro static
virgo virgo::operator,(virgo & c) {
  switch(c.i) {
    case 0 : i++;
             puts("uno!");
             break;
    case 1 : i++;
             puts("due!");
             break;
    case 2 : i++;
             puts("tre!");
             break;
    default: i = 0;
             break;
  }
  return c;
}
void Attesa(char *);
int main(int argc, char* argv[]) {
  virgo uno, due, tre, quattro;
  // quattro ha il valore di tre (con i = 1)
  quattro = (uno, due, tre, tre, tre, tre);
  Attesa("terminare");
  return 0;
}
void Attesa(char * str) {
  cout << "\n\n\tPremere return per " << str;
  cin.get();
}
/*
 * In una sequenza di espressioni separate da una virgola, le espressioni sono
 * valutate (inclusi gli effetti collaterali) da sinistra verso destra.
 * Tutti i valori risultanti dalla valutazione delle espressioni sono scartati
 * tranne il valore finale che è il risultato dell'intera sequenza; quindi tutte
 * le espressioni, tranne l'ultima, sono usate solo per i loro effetti
 * collaterali.
 * Quando l'espressione alla sinistra della virgola é un oggetto, o produce un
 * oggetto, l'operatore virgola pu• essere overloaded per generare un effetto
 * collaterale.
 * Le parentesi sono necessarie perchè, l'operatore = ha precedenza sullo
 * operatore virgola.
 */
```

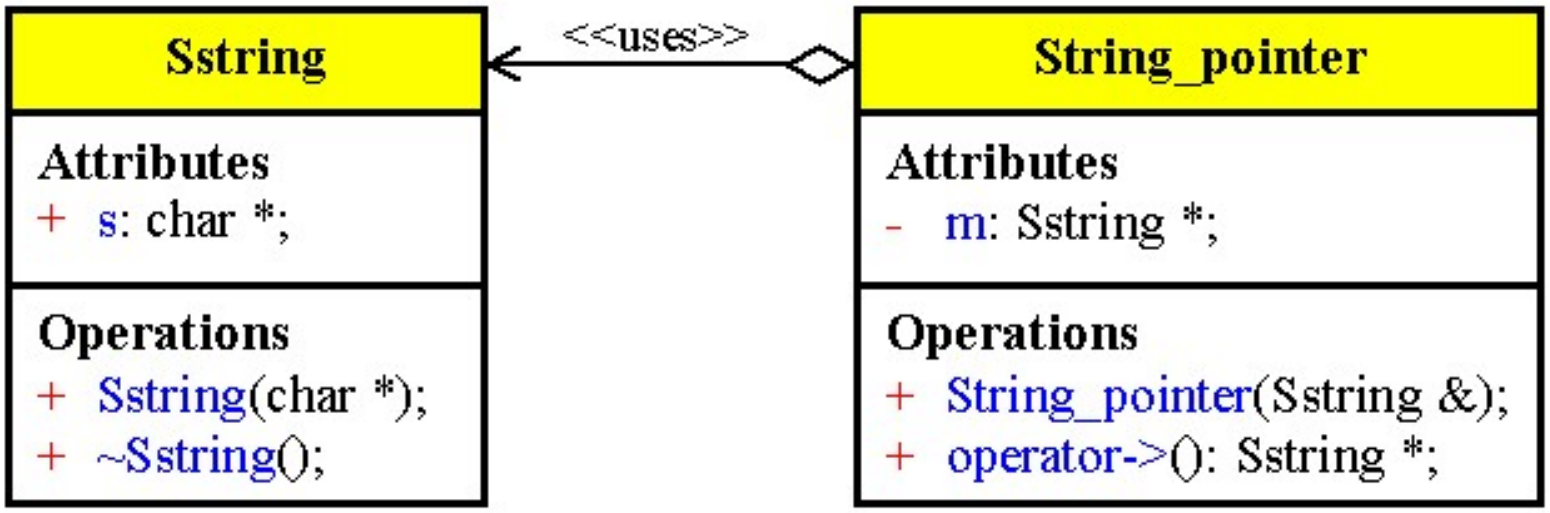
Diagramma U. M. L. delle classi



Ritorna

```
/* Smart.cpp
 * Esempio di "Puntatore intelligente"
 */
#pragma hdrstop
#include <condefs.h>
#include <iostream.h>
#include <string.h>
//---------------------------------------------------------------------------
#pragma argsused
struct Sstring {
  char *s;
  Sstring(char *S) : s(strdup(S)) {}
  ~Sstring() { free(s); }
};
class String_pointer {
  Sstring *m;
public:
  String_pointer(Sstring & S) : m(&S) {}
  Sstring * operator->() {
    // Esegue un test
    return m;
  }
};
void Attesa(char *);
int main(void) {
  Sstring st("Ciao, mondo!\n");
  String_pointer X(st);
  cout << X->s;
  Attesa("terminare");
  return 0;
}
void Attesa(char * str) {
  cout << "\n\n\tPremere return per " << str;
  cin.get();
}
/*
 * L'operatore unario operator->() puo` essere definito solo come una funzione
 * membro di una classe, e non puo` essere di tipo static.
 * Un puntatore "Intelligente", e` un puntatore che e` usato su un oggetto della
 * classe, e non per puntare a un oggetto.
 * ci• significa che l'oggetto rappresenta un puntatore, che pero` e` diverso
 * da un normale puntatore.
 * Se C e` una classe con operator->() come funzione membro, ed el e` un membro
 * di tale classe o di qualche altra classe, struttura o union, allora:
 * C cobj;
 * cobj->el;
 * E` valutato come:
 * (cobj.operator->())->el;
 * Cio` significa che operator->() deve ritornare uno tra questi due elementi:
 * - Un puntatore a un oggetto della classe contenente l'elemento el.
 *   Si derefenzia il puntatore come al solito usando l'operatore -> per
 *   selezionare el
 * - Un oggetto di un'altra classe, la quale deve contenere una definizione
 *   per operator->().
 *   In questo caso, operator->() viene chiamato di nuovo per il nuovo oggetto;
 *   questo processo e` ripetuto ricorsivamente finche` il risultato e` un
 *   puntatore a un oggetto della classe contenente l'elemento el:
 * Nell'espressione X.s, viene chiamato String_pointer::operator->(), che
 * ritorna m (non s); ma il compilatore sa che puo` andare avanti, cosi` continua
 * ad estrarre s da m, che e` un puntatore a String.
 */
```

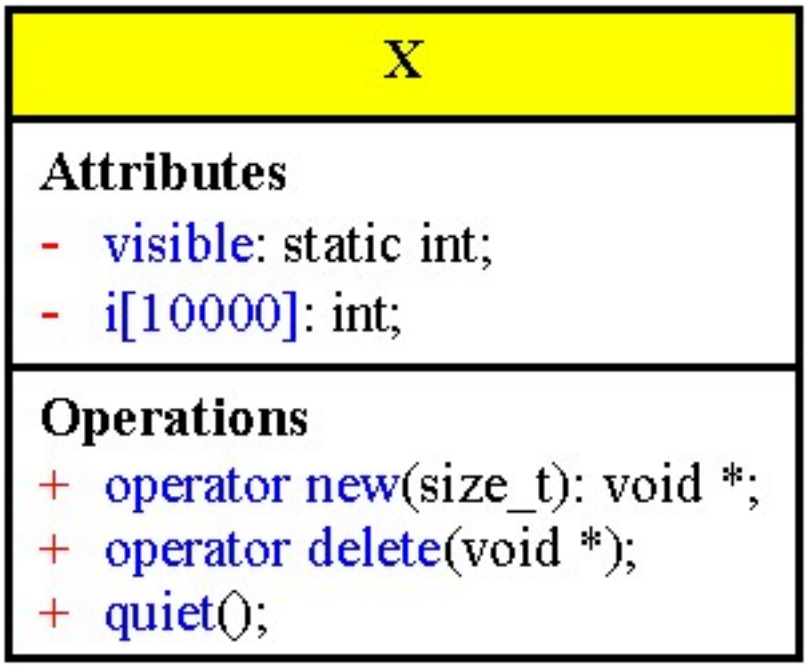
Diagramma U. M. L. delle classi



Ritorna

```
/* Opnew.cpp
 * Overloading di new e delete
 */
#pragma hdrstop
#include <condefs.h>
#include <iostream.h>
//---------------------------------------------------------------------------
#pragma argsused
#include <new.h>        // dichiara set_new_handler()
class X {
  static int visible;
  int i[10000];
public:
  void * operator new(size_t sz) {
    if(visible)
      cout << "X::operator new" << endl;
    return ::new unsigned char[sz];
  }
  void operator delete(void *dp) {
    cout << "X::operator delete" << endl;
    ::delete(dp);
  }
  static void quiet() { visible = 0; }
};

int X::visible = 1;
unsigned long count = 0;
void Attesa(char *);
void memout() {
  cerr << "Memoria libera esaurita dopo " << count << " oggetti" << endl;
  Attesa("terminare");
  exit(1);
}
void Attesa(char *);
int main(int argc, char* argv[]) {
  set_new_handler(memout);
  X *xp = new X;
  delete xp;
  X::quiet();
  while(1) {
    new X;
    ++count;
  }
  Attesa("terminare");
  return 0;
}
void Attesa(char * str) {
  cout << "\n\n\tPremere return per " << str;
  cin.get();
}
/*
 * La variabile visible, di tipo static, indica al costruttore e al distruttore
 * se devono stampare un messaggio: e' inizializzata al valore vero quando viene
 * allocata per essa la memoria, e diventa falsa quando viene chiamata la
 * funzione static quiet().
 * Nelle versioni di new e delete contenute in X, vengono chiamate le versioni
 * globali di tali funzioni, difatti si usa l'operatore :: prima del nome delle
 * funzioni.
 */
```

Diagramma U. M. L. delle classi



Operatori di conversione di tipo

Il C++ offre la possibilità di costruire tipi definiti dall'utente, che hanno forma e comportamento analoghi a qualunque tipi predefinito.
É possibile definire un operatore di conversione di tipi che possa essere applicato ai tipi definiti dall'utente.
Si può usare un costruttore ogni volta che serve convertire un tipo di dato in un altro definito dall'utente.
Non si può però usare un costruttore per convertire un dato in un tipo aritmetico, poiché non esiste un costruttore per tale tipo, in questo caso bisogna aggiungere un operatore di conversione.
Quando il compilatore si accorge che viene usato un tipo definito dall'utente, mentre è richiesto un altro tipo, chiama implicitamente l'operatore di conversione definito dall'utente.
In un operatore di conversione di tipo non è necessario indicare il valore di ritorno, il nome dell'operatore indica il valore di ritorno;
La conversione è eseguita chiamando una funzione appartenente alla classe dell'argomento.
É possibile fare in modo che la classe destinazione esegua la conversione, creando costruttori che accettano un singolo argomento, che è dell'altro tipi definito dall'utente. La classe sorgente deve concedere espliciti privilegi di accesso alla classe destinazione, oppure può fornire adeguate funzioni di accesso, in modo che il costruttore dell'oggetto della classe destinazione possa leggere i dati necessari.
Il costruttore di conversione costruisce un oggetto a partire da un altro tipo diverso.
Viene eseguito in modo implicito quando è necessaria una conversione.
Esempio:

```
class Animale {
        Animale();
        Animale(const char *nome);
};
void display_animale(const Animale &animale) {
        cout << animale.Getnome() << endl;
}
int main() {
        display_animale("Fido");        // Esegue: Animale temp("Fido");
                                                    display_animale(temp);
                                                    distrugge temp
}
```

Criteri di scelta:

1. Se si possiede la classe che deve essere convertita si può scrivere un operatore di conversione per essa
2. Se si possono estrarre sufficienti informazioni si può scrivere un costruttore che esegue la conversione di tipo per la classe destinazione.
3. Si deve usare l'operatore di conversione quando si vuole convertire un tipo definito dall'utente in un tipo predefinito, poiché non è possibile definire un costruttore per un tipo predefinito.

Problemi con troppe conversioni di tipo

Se esistono due modi per eseguire la stessa conversione di tipi il compilatore non è in grado di risolvere l'ambiguità.
Se una funzione overloaded ha più di un argomento, per il quale è definita una conversione automatica di tipo si possono avere dei problemi.

Consigli di progettazione per la conversione di tipo

1. Non si crei più di una conversione implicita di tipo per ciascun tipo definito dall'utente

Ciò non significa che vi sia una limitazione sul numero di tipi a cui si può convertire un tipo dato.
Esempio:

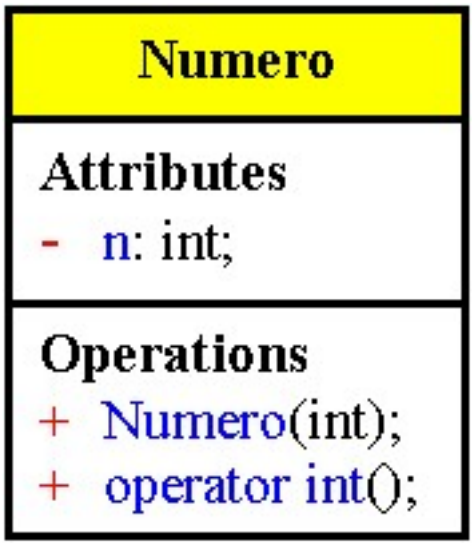
```
class uno;
class due;
class tre;
class tutti{
        long i;
public:
        tutti(int x = 0) { i = x; }
        operator uno();
        // Unica conversione implicita di tipo; le altre funzioni di conversione non possono essere
            chiamate implicitamente dal compilatore; devono essere chiamate esplicitamente dall'utente.
        due To_due();           //Converte tutti in due;
        tre To_tre();           // Converte tutti in tre
};
```

2. Un operatore di conversione dovrebbe sempre accettare un tipo complesso e generare un tipi più semplice
3. É accettabile avere mutue conversioni tra due classi, non tutte le classi sono estensioni logiche di altre classi
4. La conversione automatica di tipi dovrebbe essere usata solo se strettamente necessaria.

Ritorna

```
/* Conv_op.cpp
 * Operatore di conversione
 */
#pragma hdrstop
#include <condefs.h>
#include <iostream.h>

//---------------------------------------------------------------------------
#pragma argsused
class Numero {
public:
  Numero (int valore) { // converte da int a Numero
    n = valore;
    cout << "Costruttore di Numero" << endl;
  }
  operator int()        {       // converte da Numero a int
    cout << "Operatore di conversione" << endl;
    return n;
  }
private:
  int   n;
};
void Attesa(char *);
int main(int argc, char* argv[]) {
  int   i = 7;
  cout << "Definizione dei Numeri" << endl;
  Numero        n (12), n1 = 15;
  cout << "Chiamata ad abs" << endl;
  i = abs (n);
  cout << "Da int a numero" << endl;
  n1 = 3.14 + i;
  Attesa("terminare");
  return 0;
}
void Attesa(char * str) {
  cout << "\n\n\tPremere return per " << str;
  cin.get();
}
```

Diagramma U. M. L. delle classi

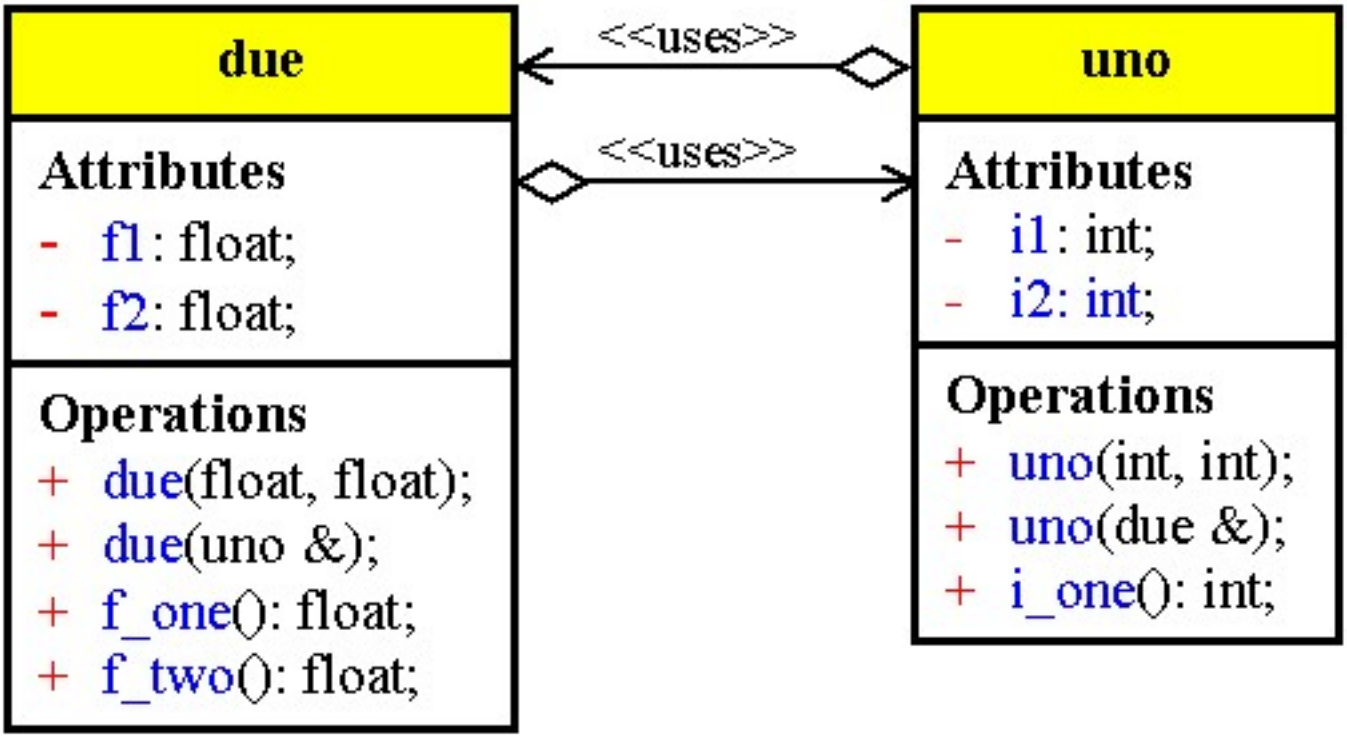


Ritorna

```
/* Constcnv.cpp
 * Costruttore con un solo argomento, per eseguire la
 * conversione di tipo.
 * Vengono mostrate due diverse alternative per permettere all'oggetto
 * destinazione di accedere ai dati privati della classe sorgente
 */
#pragma hdrstop
#include <condefs.h>
#include <iostream.h>
#include "Elem.h"
//---------------------------------------------------------------------------
USEUNIT("Elem.cpp");
//---------------------------------------------------------------------------
#pragma argsused
void funz1(due s) {
  cout << s.f_one() << endl;
}
void funz2(uno m) {
  cout << m.i_one() << endl;
}
void Attesa(char *);
int main(int argc, char* argv[]) {
  due S(5.6, 4.3);
  funz1(S);
  uno M(2, 7);
  funz2(M);
  M = S;                        // uno::uno(due &) chiamata
  S = M;                        // due::due(uno &) chiamata
  funz1(M);             // due::due(uno &) chiamata
  funz2(S);             // uno::uno(due &) chiamata
  Attesa("terminare");
  return 0;
}
void Attesa(char * str) {
  cout << "\n\n\tPremere return per " << str;
  cin.get();
}
/*
 * La classe due contiene le funzioni di accesso f_one() e f_two() che
 * forniscono alcostruttore uno::uno(due &) sufficienti informazioni
 * per costruire un oggetto uno da un oggetto due.
 * La classe uno non contiene tali funzioni di accesso; perche' il costruttore
 * due::due(uno &) contengono sufficienti informazioni per costruire un
 * oggetto due da un oggetto uno, deve essere dichiarato di tipo friend
 * nella classe uno in modo che possa accedere ai dati privati di uno
 * Regole:
 * 1) Se si possiede la classe che deve essere convertita
 *    si puo' scrivere un operatore di conversione per essa
 * 2) Se si possono estrarre sufficienti informazioni
 *    si puo' scrivere un costruttore che esegue la conversione di tipo per la
 *    classe destinazione.
 */
```

```
/* Elem.h
 */
//---------------------------------------------------------------------------
#ifndef ElemH
#define ElemH
class uno;                      // Dichiarazione della classe
class due {
public:
  due(float x1 = 0.0, float x2 = 0.0) {
    f1 = x1;
    f2 = x2;
  }
  due(uno &);   // Costruttore di conversione di tipo
  float f_one() { return f1; }  // Funzioni diaccesso
  float f_two() { return f2; }
private:
  float f1, f2;
};
class uno {                             // Definizione della classe
public:
  uno(int y1 = 0, int y2 = 0) {
    i1 = y1;
    i2 = y2;
  }
  uno(due &);   // Costruttore di conversione di tipo
  friend due::due(uno &);       // Permette l'accesso ai dati privati
  int i_one() { return i1; }
private:
  int i1, i2;
};
//---------------------------------------------------------------------------
#endif


//---------------------------------------------------------------------------
/* Elem.cpp
 */
//---------------------------------------------------------------------------
#pragma hdrstop
#include <iostream.h>
#include "Elem.h"
//---------------------------------------------------------------------------
#pragma package(smart_init)
// Questo costruttore ha il permesso di leggere i dati privati di uno, poichè
// e' una funzione amica
due::due(uno & m) {
  f1 = m.i1;                    // Conversione implicita a float
  f2 = m.i2;
  cout << "Conversione di uno in due" << endl;
}

// Questo costruttore usa le funzioni di accesso in due per leggere i dati
// privati. Non e' una funzione amica, quindi deve ottenere i dati non leggendo
// gli elementi ma in qualche altro modo
uno::uno(due & s) {
  i1 = (int)s.f_one();
  i2 = (int)s.f_two();
  cout << "Conversione di due in uno" << endl;
}
```

Diagramma U. M. L. delle classi

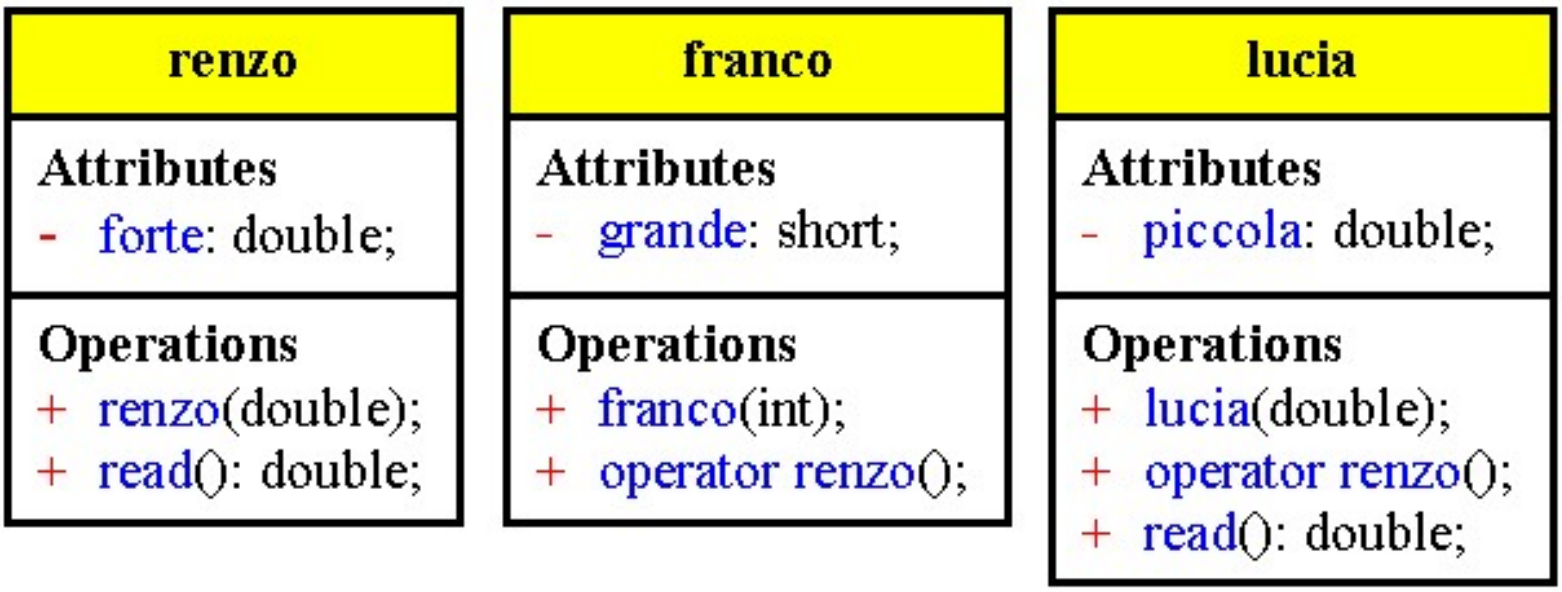


Ritorna

```
/* Ambig.cpp
 * Troppe conversioni automatiche di tipo causano ambiguita'
 */
#pragma hdrstop
#include <condefs.h>
#include <iostream.h>
#include "Amici.h"
//---------------------------------------------------------------------------
USEUNIT("Amici.cpp");
//---------------------------------------------------------------------------
#pragma argsused
void Attesa(char *);
void prova(renzo r) {
  cout << r.read() << endl;
}
void prova(lucia l) {
  cout << l.read() << endl;
}
int main(int argc, char* argv[]) {
  franco F(5);
  prova(F);
  // Si deve costruire un oggetto renzo o lucia a partire da franco?
  Attesa("terminare");
  return 0;
}
void Attesa(char * str) {
  cout << "\n\n\tPremere return per " << str;
  cin.get();
}
/*
 * Se una funzione overloaded ha piu' di un argomento, per il quale e' definita
 * una conversione automatica di tipo, vi possono essere dei problemi
 * La soluzione e' definire una sola conversione di tipo implicita e per le
 * altre l'utente deve chiamarle esplicitamente
 */
```

```
/* Amici.h
 */
//---------------------------------------------------------------------------
#ifndef AmiciH
#define AmiciH
class lucia;

class renzo {
  double forte;
public:
  renzo(double f = 0.0) { forte = f; }
  //renzo(lucia);               // Conversione di tipo da lucia a renzo
  double read() { return forte; }
};

class franco {
  short grande;
public:
  franco(int i = 0) { grande = i; }
  operator renzo();
  //operator lucia();
};

class lucia {
  double piccola;
public:
  lucia(double f = 0.0) { piccola = f; }
  // Un secondo modo per convertire lucia in renzo
  operator renzo();
  double read() { return piccola; }
};
//---------------------------------------------------------------------------
#endif


//---------------------------------------------------------------------------
/* Amici.cpp
 */
//---------------------------------------------------------------------------
#pragma hdrstop
#include <iostream.h>
#include "Amici.h"
//---------------------------------------------------------------------------
#pragma package(smart_init)
franco::operator renzo() {
  cout << "Conversione di franco in renzo" << endl;
  return renzo((double) grande);
}
lucia::operator renzo() {
  cout << "Conversione di lucia in renzo" << endl;
  return renzo(piccola);
}
```

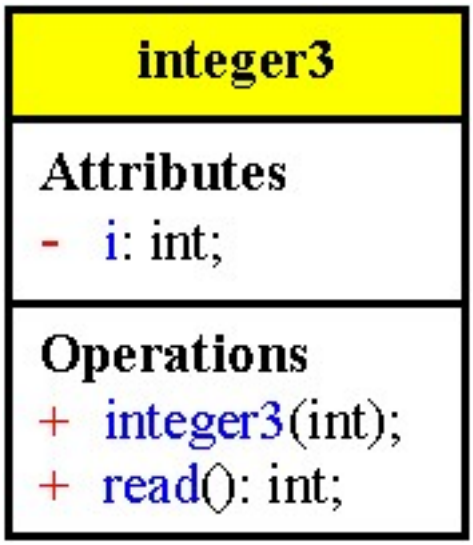
Diagramma U. M. L. delle classi



Ritorna

```
/* Succint.cpp
 * La conversione implicita di tipo puo' eliminare codice
 * ripetitivo
 */
#pragma hdrstop
#include <condefs.h>
#include <iostream.h>

//---------------------------------------------------------------------------
#pragma argsused
class integer3 {
public:
  // Creando un costruttore che ha come argomento un int il traduttore puo'
  // eseguire la conversione implicita di tipo ...
  integer3(int ii = 0) { i = ii; }
  // ... in modo che l'unica definizione richiesta è operator+()
  friend integer3 operator+(integer3, integer3);
  int read() { return i; }
private:
  int i;
};

integer3 operator+(integer3 x, integer3 y) {
  return integer3(x.i + y.i);
}
void Attesa(char *);
int main(void) {
  integer3 A(10), B(20), C;
  C = A + B;                                          // Legale
  cout << "C = A + B  =  " << C.read() << endl;
  C = A + 4;                                          // Legale
  cout << "C = A + 4  =  " << C.read() << endl;
  C = 5 + A;                                          // Legale
  cout << "C = 5 + A  =  " << C.read() << endl;
  Attesa("terminare");
  return 0;
}
void Attesa(char * str) {
  cout << "\n\n\tPremere return per " << str;
  cin.get();
}
/*
 * Si utilizzi una funzione membro se non vi sono altre necessita' oppure se si
 * vuole forzare una particolare sintassi.
 * Se la sintassi richiede che gli argomenti siano indipendenti, si utilizzi
 * una funzione amica e possibilmente un operatore di conversione implicita di
 * tipo per compattare il codice
 */
```

Diagramma U. M. L. delle classi



4° Modulo

- Reimpiego del codice
  - Composizione
  - Ereditarietà
    - Inizializzazione
    - Multipla
      - Virtuale
- Polimorfismo
  - VTable
  - Classi astratte
- Contenitori
- Template
  - Argomenti
  - Membri static
  - Modelli di funzione
- Puntatori a membro
- Funtore
- Eccezioni
  - Throw
  - Try-catch
  - Nei costruttori
- RTTI
  - Typeinfo
  - Typeid
- Namespace
- Cast
  - Dynamic_cast
  - Static_cast
  - Const_cast
  - Reinterpret_cast
- STL
  - Contenitori
  - Iteratori
  - Algoritmi
  - Allocatori

Reimpiego del codice

Le classi possono essere riutilizzate in due modi.

1. Per composizione - creando all'interno di un'altra classe oggetti membri di altre classi, un oggetto può essere composto da altri oggetti.

Relazione d'uso: tra i membri della classe A c'è un oggetto della classe B
Esempio un edificio è composto da porte, finestre, stanze

2. Ereditarietà - creando oggetti che sono specie particolari di altri oggetti, una classe derivata eredita le caratteristiche di una classe base.

Relazione di ereditarietà: si dichiara che la classe A deriva dalla classe B; ogni oggetto A è anche un oggetto B
Esempio: un grattacielo è una specie di edificio.

Composizione

Per utilizzare la composizione è sufficiente dichiarare un oggetto di una classe come membro di una nuova classe.
Quando si definisce un normale oggetto viene chiamato il costruttore dell'oggetto con una lista di argomenti dopo l'identificatore.
La dichiarazione dell'oggetto membro indica al compilatore di aggiungere spazio per l'oggetto membro in ogni istanza della nuova classe.
Il costruttore non può essere chiamato fino a quando lo spazio non è allocato, cioè fino a quando non si crea una istanza della nuova classe
Per controllare la costruzione di tutti i sotto-oggetti si utilizza la lista di inizializzazione del costruttore.
Quando si entra nel corpo del costruttore della nuova classe si deve essere sicuri che tutti gli oggetti membri siano stati inizializzati.
I costruttori di tutti gli oggetti membri devono essere chiamati prima del costruttore della nuova classe.
La lista di inizializzazione del costruttore mostra le chiamate da fare all'esterno del corpo del costruttore della nuova classe, dopo la lista degli argomenti e prima della parentesi graffa aperta
Esempio:

```
class table {
        int tbl[];
        int size;
public:
        table(int sz = 15);
        ~table();
};
class classdef {
        table membri;
        int nmembri;
public:
        classdef(int size);
        ~classdef();
};
classdef :: classdef(int size) : membri(size) {
        nmembri = size;
}
```

Ereditarietà

L'ereditarietà definisce anche nuove relazioni, quando si deriva una nuova classe da una classe base:

la nuova classe non acquisisce semplicemente le caratteristiche della classe base
le istanze della nuova classe sono istanze della classe base
le funzioni della classe base continuano a funzionare anche per gli oggetti della classe derivata
tutte le classi derivate hanno lo stesso insieme di funzioni, interfaccia, della classe base.

Con l'ereditarietà è possibile:

1. Astrarre affinità di comportamenti tra oggetti simili attraverso il meccanismo di generalizzazione
2. Raggiungere un maggior dettaglio di descrizione attraverso la specializzazione di classi a partire da classi più globali

Tipi di ereditarietà:

Pubblica	lascia immutate le categorie dei membri
Protetta	i membri pubblici diventano protected
Privata	i membri pubblici e protetti diventano privati.

L'ereditarietà privata e protetta offrono un modo di derivare solo una parte dell'interfaccia pubblica di una classe.

Scrittura di programmi espandibili

L'uso dell'ereditarietà per la modifica di classi esistenti rappresenta un metodo molto potente per lo sviluppo incrementale del codice, anche per il fatto che è possibile derivare una classe anche senza la disponibilità del codice sorgente relativo alle funzioni della classe.
Classificazione: fornire descrizioni chiare e precise di oggetti in termini di somiglianza e differenza::
Metodo di classificazione:

esame di un oggetto noto
eliminazione di alcune sue caratteristiche
aggiunta delle caratteristiche specifiche

Descrivere l'oggetto astratto che possiede tutte le caratteristiche comuni e poi esplicitare gli aspetti specifici delle diverse realizzazioni.
Tutte le sottoclassi hanno a disposizione le funzioni della classe base a meno che esse non siano state nascoste ereditando privatamente dalla classe base.

Vantaggi:

Uno dei principali vantaggi della programmazione orientata agli oggetti consiste nella progettazione dei programmi; molti problemi possono essere rappresentati secondo una struttura gerarchica ad albero e l'ereditarietà rappresenta un modo per classificare i concetti.
L'albero gerarchico mostra quali concetti hanno caratteristiche comuni.
In tal modo si possono riusare parti di codice riferendosi solamente alle indicazioni presenti nell'interfaccia.
Inoltre si ha la possibilità di modificare liberamente le classi che si riusano senza avere a disposizione il codice sorgente.
Creare programmi espandibili aggiungendo funzionalità a moduli già realizzati.

Ordine di inizializzazione

Quando una classe viene derivata da un'altra classe base deve essere inizializzata prima di entrare nel corpo del costruttore della classe derivata
Il costruttore della classe base deve essere chiamato prima di entrare nel corpo del costruttore della classe derivata.
Nel caso in cui l'oggetto sia un conglomerato di altri oggetti
i costruttori vengono chiamati nell'ordine:

1. il costruttore della classe base
2. i costruttori degli oggetti membri
3. il costruttore della classe derivata

i distruttori sono chiamati nell'ordine inverso

1. il distruttore della classe derivata
2. i distruttori delle classi membro
3. il distruttore della classe base

Un oggetto membro di una classe, che è anche istanza di una seconda classe, deve essere inizializzato in modo speciale: quando si entra nel corpo del costruttore della classe che contiene l'oggetto membro, quest'ultimo deve essere già inizializzato.
Regole:

Le classi base sono inizializzate per prime, nell'ordine in cui appaiono nella lista delle classi base
Vengono inizializzati gli oggetti membro nell'ordine in cui appaiono nella dichiarazione della classe.
Viene chiamato il codice del costruttore.

Se una delle classi base ha delle classi basi o degli oggetti membro la stessa regola è applicata ricorsivamente.

Ereditarietà multipla

Possibilità di dichiarare che una classe deriva le sue caratteristiche da due o più classi, sommando tra loro tutte quelle delle classi base.
Vantaggi:

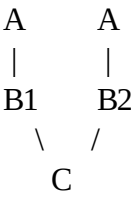
Maggiore flessibilità, semplicità nella definizione di nuove classi a partire da altre già esistenti.
Permette di creare gerarchie di classi senza che il legame classe-gerarchia sia univoco, permettendo la progettazione di applicazioni più flessibili.

Derivazione virtuale

Volendo creare funzioni e programmi che manipolano oggetti di una classe base, in modo che tali funzioni e programmi siano espandibili mediante l'ereditarietà si deve affrontare il problema in cui si vuole utilizzare l'interfaccia della classe base, ma con le diverse implementazioni delle funzioni create nelle classi derivate.
Nella situazione:

```
class A;
class B1 : public A;
class B2 : public A;
class C : public B1, public B2;
```

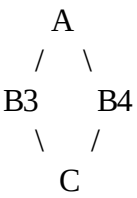
Il compilatore genera due copie della classe base ripetuta, indipendenti tra loro creando la struttura:



La chiamata di una funzione membro di A all'interno di C risulta ambigua.
Per questo bisogna premettere a questa chiamata il nome della classe da cui deriva direttamente
Esempio: B1::dato e non dato oppure A::dato perché ambigue.
Si può forzare il compilatore a generare un'unica copia della classe base ripetuta:

```
class A;
class B3 : virtual public A;
class B4 : virtual public A;
class D : public B3, public B4;
```

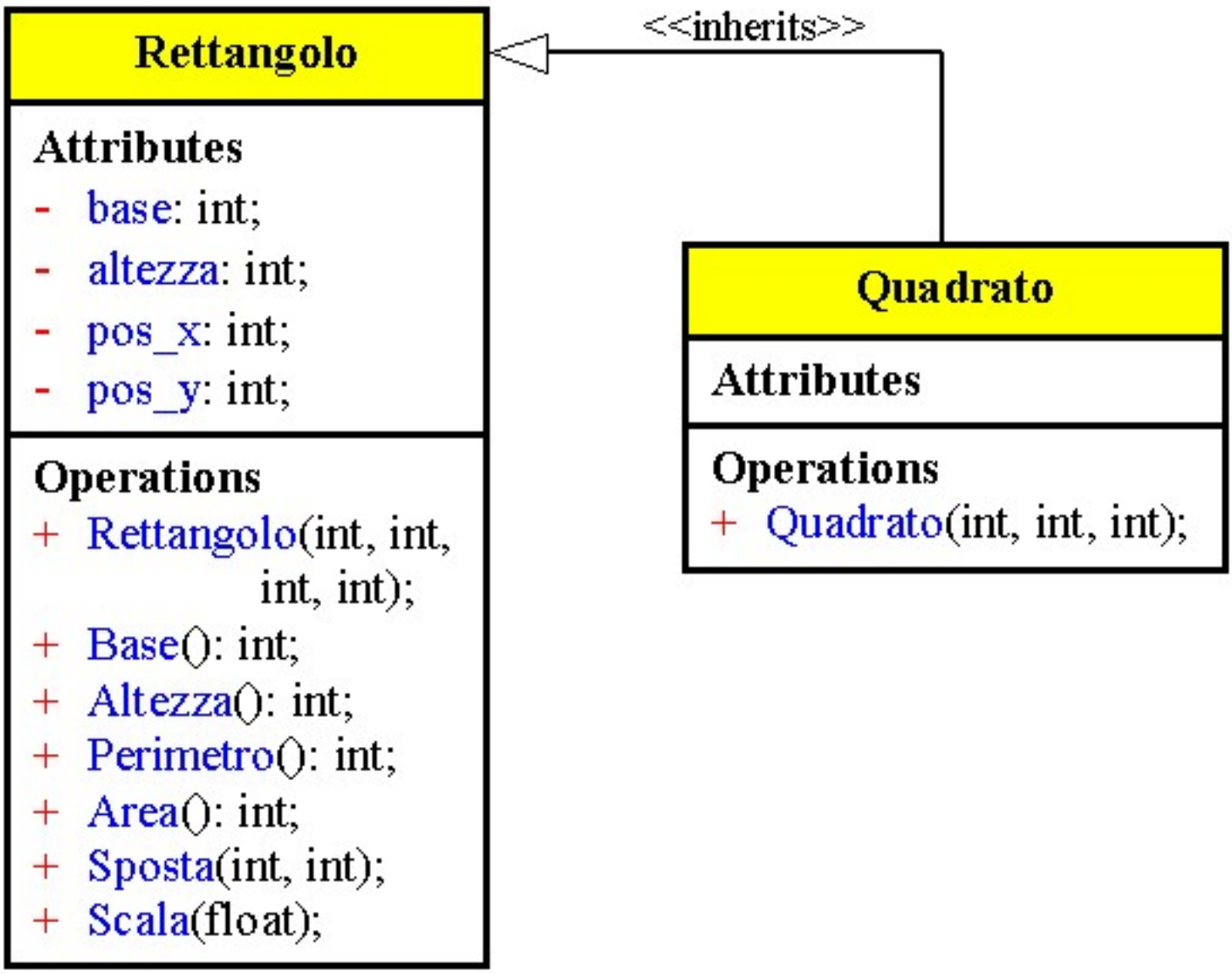
Il compilatore genera una sola copia di A secondo la struttura:



Ritorna

```
/* Rectang.h
 * definizione della classe Rettangolo
 */
#include <iostream.h>

class Rettangolo {
private:
public:
  // costruttore
  Rettangolo (int b, int a, int x = 0, int y = 0) {
    base = b;
    altezza = a;
    pos_x = x;
    pos_y = y;
  }
  int   Base () {
    return base;
  }
  int   Altezza () {
    return altezza;
  }
  int   Perimetro () {
    return ( (base + altezza) * 2 );
  }
  int   Area () {
    return ( base * altezza );
  }
  // sposta il rettangolo
  void  Sposta (int delta_x, int delta_y) {
    pos_x += delta_x;
    pos_y += delta_y;
  }
  // modifica la scala
  void  Scala (float scala) {
    base *= scala;
    altezza *= scala;
  }
private:
  // base, altezza e posizione dell'angolo inferiore sinistro
  int base, altezza;
  int pos_x, pos_y;
};
ostream& operator << (ostream& s, Rettangolo r) {
  s << "Rettangolo: base = " << r.Base() << ", altezza = "
    << r.Altezza() << endl;
  s << "\tArea = " << r.Area() << ", perimetro = "<< r.Perimetro() << endl;
  return s;
}
//---------------------------------------------------------------------------
/* Square1.cpp
 * derivazione della classe Quadrato dalla classe Rettangolo
 * prima versione)
 */
#pragma hdrstop
#include <condefs.h>
#include <iostream.h>
#include "Rectang.h"
//---------------------------------------------------------------------------
#pragma argsused
class Quadrato : public Rettangolo {
public:
  Quadrato (int lato, int pos_x = 0, int pos_y = 0):
    Rettangolo (lato, lato, pos_x, pos_y)
  { }
};
void Attesa(char *);
int main(void) {
  Quadrato quad (15, 12, 18);
  cout << "Il perimetro del quadrato e' " << quad.Perimetro() << endl;
  Attesa("terminare");
  return 0;
}
void Attesa(char * str) {
  cout << "\n\n\tPremere return per " << str;
  cin.get();
}
```

Diagramma U. M. L. delle classi

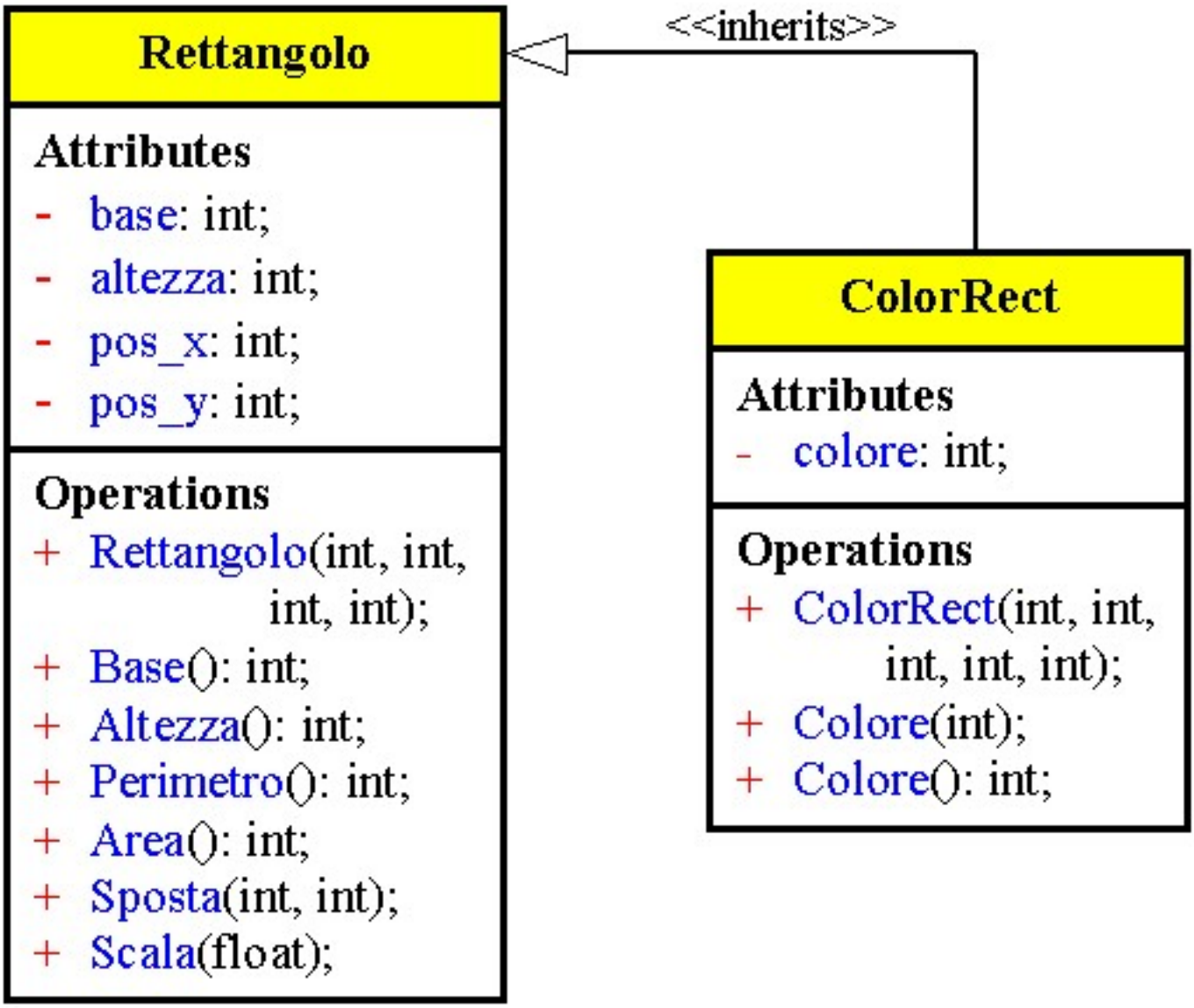


Ritorna

```
/* Rectcol.h
 * definizione di una classe di rettangoli colorati
 * a partire dalla classe Rettangolo
 */
#include "Rectang.h"

class ColorRect: public Rettangolo {
public:
  ColorRect (int b, int a, int c = 4, int x = 0, int y = 0):
    Rettangolo (b, a, x, y),
    colore (c)
  { }
  void Colore (int c) {
    colore = c;
  }
  int Colore() {
    return colore;
  }
private:
  int   colore; // l'int corrisponde a un colore
};

ostream& operator << (ostream& s, ColorRect cr) {
  // mostra il ColorRect come se fosse un Rettangolo
  s << (Rettangolo)cr;
  // mostra i dati specifici (il colore)
  s << "\tcolore = " << cr.Colore() << endl;
  return s;
}
//---------------------------------------------------------------------------
/* Rectcol.cpp
 * esempio di uso della classe ColorRect
 */
#pragma hdrstop
#include <condefs.h>
#include <iostream.h>
#include "Rectcol.h"
//---------------------------------------------------------------------------
#pragma argsused
void Attesa(char *);
int main(int argc, char* argv[]) {
  // definisce un rettangolo colorato
  ColorRect cr(10, 15, 5);
  cout << "Questo rettangolo e' colorato" << endl;
  cout << cr << endl;
  // il colore cambia
  cr.Colore(8);
  cout << "Il colore e' cambiato" << endl;
  cout << cr << endl;
  Attesa("terminare");
  return 0;
}
void Attesa(char * str) {
  cout << "\n\n\tPremere return per " << str;
  cin.get();
}
```

```
/* Rectang.h
 * definizione della classe Rettangolo
 */
#include <iostream.h>

class Rettangolo {
private:
public:
  // costruttore
  Rettangolo (int b, int a, int x = 0, int y = 0) {
    base = b;
    altezza = a;
    pos_x = x;
    pos_y = y;
  }
  int   Base () {
    return base;
  }
  int   Altezza () {
    return altezza;
  }
  int   Perimetro () {
    return ( (base + altezza) * 2 );
  }
  int   Area () {
    return ( base * altezza );
  }
  // sposta il rettangolo
  void  Sposta (int delta_x, int delta_y) {
    pos_x += delta_x;
    pos_y += delta_y;
  }
  // modifica la scala
  void  Scala (float scala) {
    base *= scala;
    altezza *= scala;
  }
private:
  // base, altezza e posizione dell'angolo inferiore sinistro
  int base, altezza;
  int pos_x, pos_y;
};
ostream& operator << (ostream& s, Rettangolo r) {
  s << "Rettangolo: base = " << r.Base() << ", altezza = "
    << r.Altezza() << endl;
  s << "\tArea = " << r.Area() << ", perimetro = "<< r.Perimetro() << endl;
  return s;
}
```

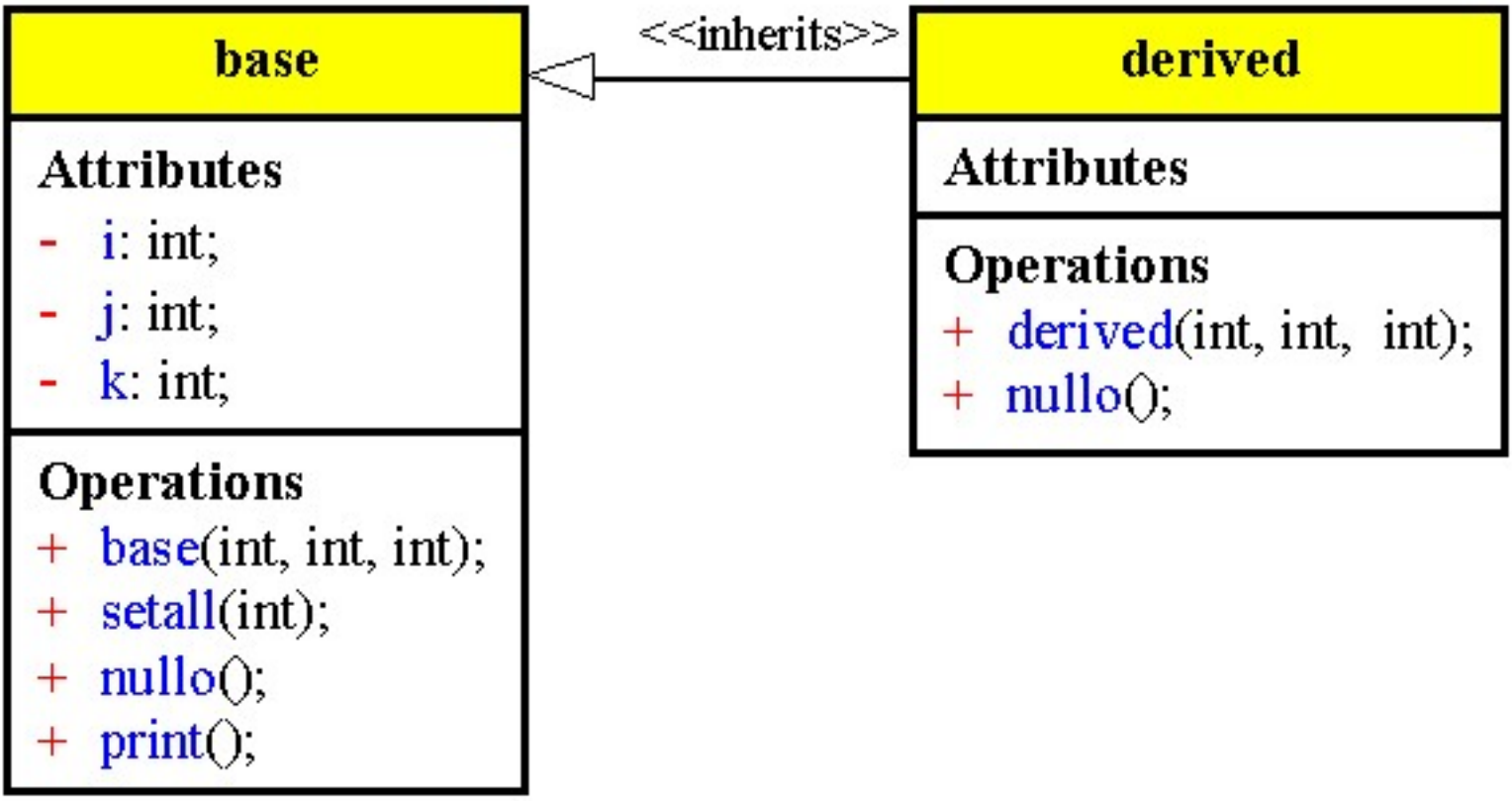
Diagramma U. M. L. delle classi



Ritorna

```
/* Hide1.cpp
 * Ridefinizione di una funzione della classe base per
 * nasconderla
 */
#pragma hdrstop
#include <condefs.h>
#include <iostream.h>

//---------------------------------------------------------------------------
#pragma argsused
class base {
public:
  base(int ii = 0, int jj = 0, int kk = 0) {
    i = ii;
    j = jj;
    k = kk;
  }
  void setall(int val) {
    i = j = k = val;
  }
  void nullo() {
    i = j = k = 0;
  }
  void print() {
    cout << "i = " << i << " j = " << j << " k = " << k << endl;
  }
private:
  int i, j, k;
};

class derived : public base {
public:
  derived(int x, int y, int z) : base(x, y, z) {}
  // setall() e` automaticamente disponibile in derived
  // Per impedire all'utente di chiamare la funzione nullo(), la si puo`
  // ridefinire per generare un messaggio di errore
  void nullo() {
    cout << "nullo() non disponibile" << endl;
  }
};
void Attesa(char *);
int main(void) {
  base A;
  A.setall(1);
  A.print();
  A.nullo();
  A.print();
  derived B(1, 2, 3);
  B.print();
  B.setall(2);
  B.print();
  B.nullo();            // Provoca un messaggio di errore a tempo di esecuzione
  Attesa("terminare");
  return 0;
}
void Attesa(char * str) {
  cout << "\n\n\tPremere return per " << str;
  cin.get();
}
/*
 * Per evitare che l'utente possa usare alcune funzioni della classe base
 * si puo` ridefinire la funzione nella classe derivata, in modo che non
 * compia nessuna azione dannosa, o in modo che stampi un messaggio di errore
 */
```

Diagramma U. M. L. delle classi

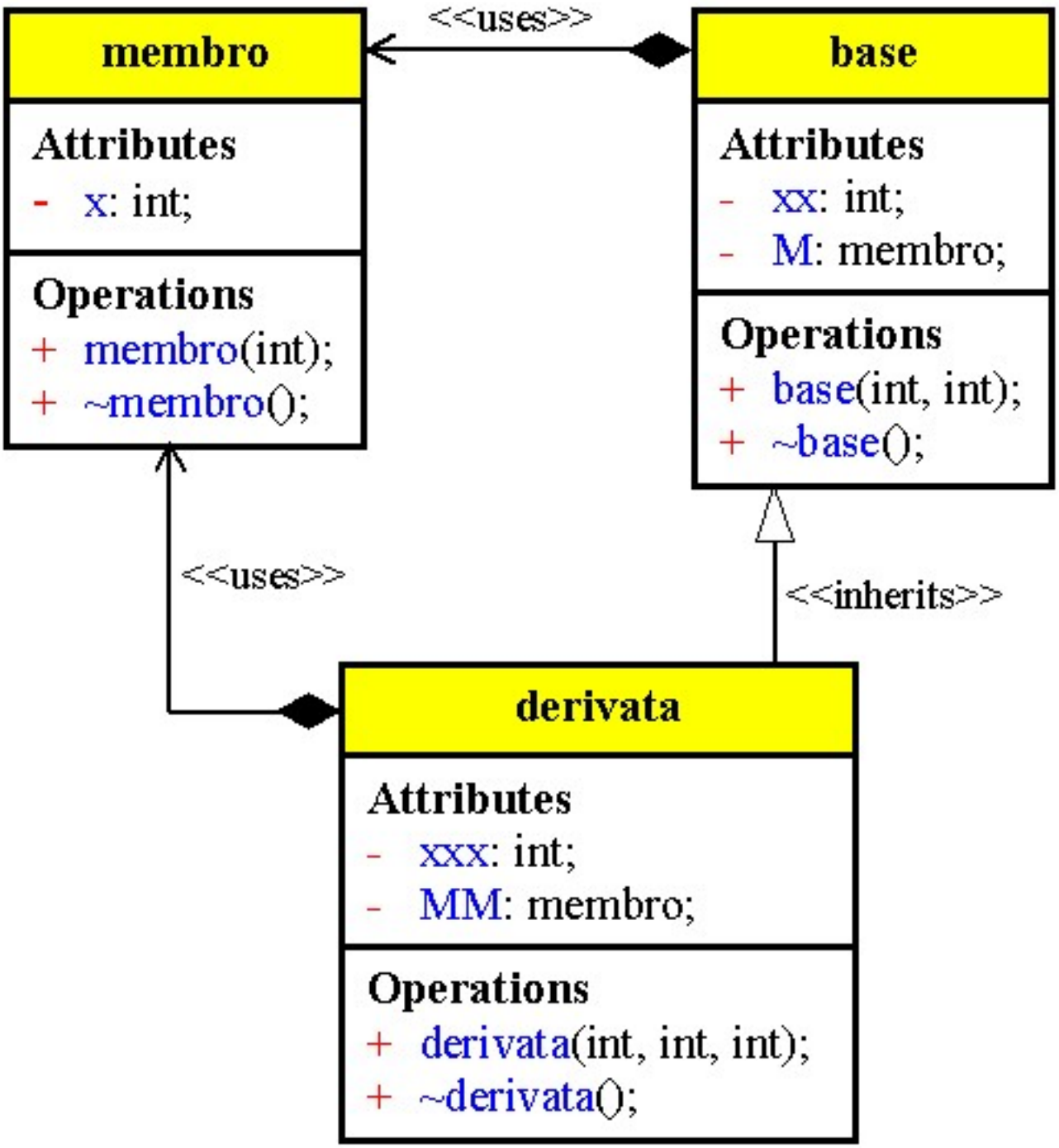


Ritorna

```
/* Order.cpp
 * Ordine delle chiamate del costruttore e del distruttore per
 * oggetti ereditati e oggetti contenenti oggetti membri.
 */
#pragma hdrstop
#include <condefs.h>
#include <iostream.h>

//---------------------------------------------------------------------------
#pragma argsused
class membro {                  // Da usare come oggetto membro
public:
  membro(int i) {
    cout << "Costruttore membro chiamato con argomento " << i << endl;
    x = i;
  }
  ~membro() {
    cout << "Distruttore membro chiamato, x = " << x << endl;
  }
private:
  int x;                                // Per ricordare che oggetto é'
};

class base {                            // Da usare come una classe base
public:
  // inizializzazione di un oggetto membro
  base(int a, int b) : M(b) {
    cout << "Costruttore base chiamato con argomenti " << a << ", " << b
         << endl;
    xx = a;
  }
  ~base() {
    cout << "Distruttore base chiamato, xx = " << xx << endl;
  }
private:
  int xx;                       // Per ricordare che oggetto e'
  membro M;                     // Oggetto membro
};

// Si osservi attentamente cosa succede con l'ereditarieta', e con un altro
// oggetto membro
class derivata : public base {          // <-- Ereditarieta'
public:
    // Ordine di chiamata dei costruttori della classe base e delle funzioni
    // membro. Sono indicati prima del corpo della classe, per ricordare che
    // sono chiamati prima del costruttore della classe.
    // Questa e' la "lista di inizializzazione del costruttore"
    derivata(int a, int b, int c) : base(a, b), MM(c) {
    cout << "Costruttore derivata chiamato con argomenti " << a << ", "
         << b << ", " << c << endl;
    xxx = a;
  }
  ~derivata() {
    cout << "Distruttore derivata chiamato, xxx = " << xxx << endl;
  }
private:
  int xxx;                              // Per ricordare che oggetto e'
  membro MM;
};
void Attesa(char *);
int main(int argc, char* argv[]) {
  {
    cout << "Creazione di derivata X(1, 2, 3);" << endl;
    derivata X(1, 2, 3);
    cout << "derivata X(1, 2, 3) non più visibile" << endl;
  }             // <-- distruttore chiamato
  cout << "Creazione di derivata *dp = new derivata(4, 5, 6); " << endl;
  derivata *dp = new derivata(4, 5, 6);
  cout << "Chiamata di delete dp; " << endl;
  delete dp;
  Attesa("terminare");
  return 0;
}
void Attesa(char * str) {
  cout << "\n\n\tPremere return per " << str;
  cin.get();
```

```
}
/*
 * Poiche' il C++ crea dinamicamente gli oggetti, e non alloca semplicemente la
 * memoria, i costruttori devono essere chiamati in modo che new ritorni
 * l'indirizzo di un oggetto "vivo". Analogamente, il distruttore per un
 * oggetto deve essere chiamato, prima che la memoria per quell'oggetto sia
 * deallocata.
 * Nel caso in cui l'oggetto sia un conglomerato di altri oggetti (per
 * ereditarieta', o perche' contiene oggetti membri), in primo luogo viene
 * chiamato il costruttore della classe base, poi i costruttori degli oggetti
 * membri, e infine il costruttore della classe derivata.
 * Quando si usa delete per distruggere un oggetto precedentemente creato sullo
 * heap il compilatore chiama il distruttore della classe.
 * Se un oggetto e' un conglomerato di altri oggetti, i distruttori sono
 * chiamati nell'ordine inverso di chiamata dei costruttori: in primo luogo
 * viene chiamato il distruttore della classe derivata, poi i distruttori degli
 * oggetti membri, e infine il distruttore della classe base.
 * Un oggetto membro di una classe, che e' anche istanza di una seconda classe,
 * deve essere inizializzato in modo speciale: quando si entra nel corpo del
 * costruttore della classe che contiene l'oggetto membro, quest'ultimo deve
 * essere gia' inizializzato.
 * Questa e' chiamata lista di inizializzazione del costruttore.
 * Quando una classe viene derivata da un'altra classe, la classe base deve
 * essere inizializzata prima di entrare nel corpo del costruttore della classe
 * derivata.
 */
```

Diagramma U. M. L. delle classi

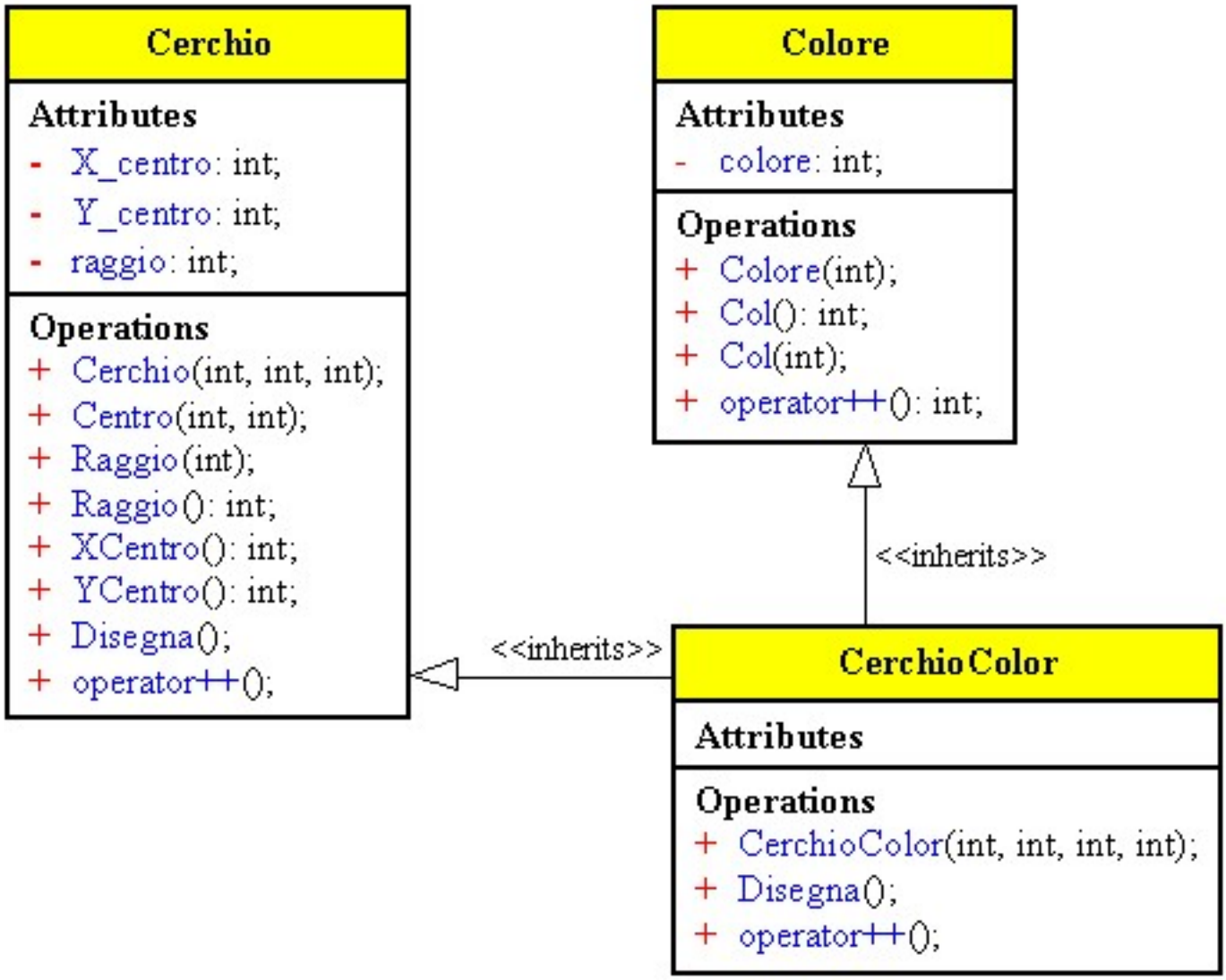


Ritorna

```
/* Circlec.h
 * definizione della classe CerchioColor, che eredita
 * sia dalla classe Cerchio che dalla classe Colore
 * (ereditarieta' multipla)
 */
class CerchioColor : public Cerchio, public Colore {
public:
  CerchioColor (int x, int y, int r = 20, int c = RED) :
    Cerchio ( x, y, r ),
    Colore ( c )
  { }
  void  Disegna () {
    Cerchio :: Disegna ();
    cout << ", il numero del colore e' " << Col();
  }
  void  operator ++ () {
    Colore :: operator++();
    Cerchio :: operator++();
  }
};

//---------------------------------------------------------------------------
/* CerchioC.cpp
 * funzione main dell'esempio CerchioColor
 */
#pragma hdrstop
#include <condefs.h>
#include <iostream.h>
#include "circle.h"
#include "color.h"
#include "circlec.h"
//---------------------------------------------------------------------------
#pragma argsused
void Attesa(char *);
int main(int argc, char* argv[]) {
  int nRag1, nRag2;
  cout << "Introdurre raggio iniziale e finale: ";
  cin >> nRag1 >> nRag2;
  for (CerchioColor cc (30, 30, nRag1, Colore::RED);
       cc.Raggio() < nRag2;
       ++cc) {
    cc.Disegna();
    cout << endl;
  }
  Attesa("terminare");
  return 0;
}
void Attesa(char * str) {
  cout << "\n\n\tPremere return per " << str;
  cin.ignore(4, '\n');
  cin.get();
}
```

```
/* Circle.h
 * definizione della classe Cerchio
 */
class Cerchio {
public:
  Cerchio ( int x, int y, int r = 20 ) :
    X_centro (x),
    Y_centro (y),
    raggio (r)
  { }
  // Modifica la posizione del centro
  void  Centro ( int x, int y ) {
    X_centro = x;
    Y_centro = y;
  }
  // modifica il raggio
  void  Raggio ( int r ) {
    raggio = r;
  }
  // funzioni che restituiscono dei dati privati
  int   Raggio () {
    return raggio;
  }
  int   XCentro () {
    return X_centro;
  }
  int   YCentro () {
    return Y_centro;
  }
  // descrive il cerchio (dovrebbe disegnarlo in grafica)
  void  Disegna () {
    cout << "Cerchio in (" << X_centro << ", " << Y_centro
         << "), raggio = " << raggio;
  }
  // incrementa dimensione e posizione
  void  operator ++ () {
    raggio ++;
    X_centro ++;
    Y_centro ++;
  }
  // i dati privati sono le coordinate X e Y del centro e il raggio
private:
  int   X_centro, Y_centro;
  int   raggio;
};

/* Color.h
 * definizione della classe Colore
 */
class Colore {
  int colore;
public:
  // definisce alcune costanti per i colori:
  enum {BLACK, RED, BLUE, GREEN, YELLOW, WHITE};
  Colore (int c = BLACK) :
    colore (c)
  { }
  // funzioni per modificare e leggere il dato privato
  int   Col () {
    return colore;
  }
  void  Col (int c) {
    colore = c;
  }
  // operatore di incremento (che opera in modulo)
  int   operator++ () {
    colore = (colore + 1) % (WHITE + 1);
    return colore;
  }
};
```

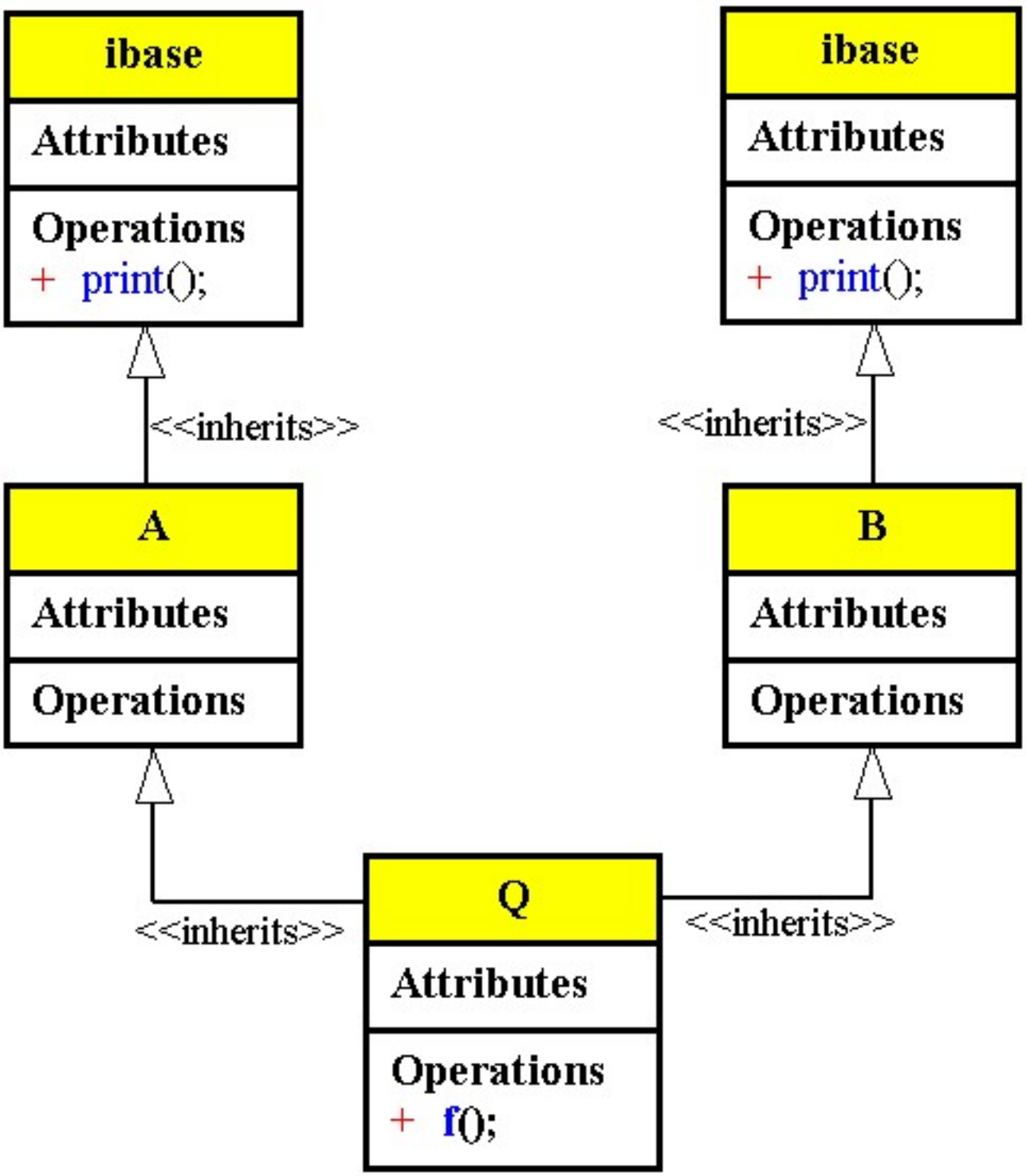
Diagramma U. M. L. delle classi



Ritorna

```
/* Ibase2.cpp
 * Eliminazione della ambiguita` dell'ereditarieta` multipla
 * per mezzo dell'utilizzo delle classi base virtuali
 */
#pragma hdrstop
#include <condefs.h>
#include <iostream.h>

//---------------------------------------------------------------------------
#pragma argsused
class ibase {
public:
  void print() {
    cout << "Sono la classe ibase" << endl;
  }
};
class A : virtual public ibase {};
class B : virtual public ibase {};
class Q : public A, public B {
public:
  void f() { print(); }         // Non ambiguo
};
void Attesa(char *);
int main(int argc, char* argv[]) {
  Q q;
  cout << "q.f()         ";
  q.f();
  Q *qp = &q;
  cout << "qp->f()       ";
  qp->f();
  ibase *ip = qp;                                  // Nessun problema
  cout << "ip->print()   ";
  ip->print();
  Attesa("terminare");
  return 0;
}
void Attesa(char * str) {
  cout << "\n\n\tPremere return per " << str;

  cin.get();
}
/*
 * E` possibile creare una classe base virtuale, caratterizzata dalla parola
 * chiave virtual; in questo modo ogni classe che eredita in maniera multipla
 * due classi contenenti la stessa classe base virtuale contiene un solo
 * sotto-oggetto, eliminando cosi` l'ambiguita` sia delle chiamate di funzioni
 * sia dello spazio di memoria per tale oggetto.
 * La classe base virtuale e` implementata come un puntatore nela classe
 * derivata; il sotto-oggetto non viene inserito direttamente nella classe
 * derivata.
 * Se in una classe derivata mediante ereditarieta` multipla, vengono ereditate
 * piu classi base della stessa radice, esse contengono soltanto puntatori a
 * un singolo sotto-oggetto della classe radice.
 */
```

Diagramma U. M. L. delle classi



Polimorfismo

Polimorfismo = molte forme, una entità si riferisce in esecuzione a istanze di classi diverse.
Binding = legame, collegamento tra una entità e le sue caratteristiche; per le funzioni è il legame tra la chiamata della funzione e il codice effettivamente eseguito.
La risoluzione di una chiamata di funzione a tempo di esecuzione è detta static binding; il compilatore fissa univocamente quale parte di codice deve essere eseguita per ogni chiamata di funzione.
Per implementare un'interfaccia comune con differenti implementazioni per le funzioni membro, la risoluzione delle funzioni deve essere ritardata a tempo di esecuzione dynamic binding
Invece di collocare l'indirizzo del codice della funzione da eseguire all'interno dell'istruzione di chiamata, il compilatore aggiunge l'indirizzo ad una tabella dotata di un ingresso per ogni ridefinizione di funzione relativa alla funzione nelle classi derivate.
Quando la funzione viene chiamata avviene una ricerca nella tabella per ottenere l'indirizzo del codice da eseguire
Per effettuare il dynamic binding di una funzione in una classe si dichiara la funzione con la parola chiave virtual.
Per ridefinire una funzione virtuale in una classe derivata bisogna fornire una funzione che corrisponde esattamente alla precedente, stessi parametri in numero e tipo altrimenti il compilatore la interpreta come overloading.
Le funzioni virtuali consentono di dichiarare funzioni in una classe base che possa essere ridefinita in ogni classe derivata.
La parola chiave virtual indica che la funzione può presentare versioni diverse e che il reperimento della versione adeguata per ogni chiamata è compito del compilatore.
Il tipo della funzione viene dichiarato nella classe base e non può essere ridichiarato in una classe derivata.
I prototipi per la versione di una funzione virtuale della classe base e per tutte le versioni delle classi derivate devono essere identici, altrimenti non si avrebbe una interfaccia comune.
Non si possono avere costruttori virtuali, ma è possibile avere distruttori virtuali.
Si possono anche avere operatori overloaded virtuali.
Con l’ereditarietà non è possibile rimuovere funzioni, e quindi le funzioni che si possono chiamare nella classe base devono esistere anche nelle classi derivate, e quindi è sempre possibile chiamare le funzioni della classe base (UPCAST conversione verso l’alto).
Una conversione verso il basso (downcast), invece, non è necessariamente sicura poiché il compilatore non può sapere a quale sottoclasse ci si riferisce.
Solo inserendo ulteriori informazioni, da ottenere a tempo di esecuzione, è possibile sapere di che oggetto si tratta; questa tecnica viene chiamata controllo di tipo a tempo di esecuzione.
La chiave della programmazione orientata agli oggetti consiste nell’inviare un messaggio ad un oggetto, e lasciare che l’oggetto lo gestisca.

Meccanismo delle funzioni virtuali:

Quando si dichiara virtuale una funzione di una classe, il compilatore aggiunge alla classe, in maniera trasparente, un oggetto, chiamato VPTR, che è un puntatore a una tabella di puntatori a funzioni chiamata VTABLE, che contiene a sua volta i puntatori a tutte le funzioni della classe o delle classi derivate che sono dichiarate virtuali.
In un programma C++ quando viene effettuata una chiamata a una funzione virtuale, il compilatore genera del codice per trattare VPTR come indirizzo iniziale di un array di puntatori a funzioni; questo codice semplicemente esegue un'indicizzazione all'interno dell'array e chiama la funzione corrispondente.
Con ciò è possibile chiamare una funzione membro virtuale mentre si tratta l'oggetto come membro della classe base, e la giusta funzione della classe derivata viene chiamata.
Consideriamo:

```
struct X{
        int i;
        virtual void f();
        virtual void g();
};
X x, *xp = &x;
xp->g();
```

Il membro i è contenuto nella struttura e il compilatore ha anche aggiunto alla struttura il puntatore VPTR, che punta a un array di puntatori a funzioni chiamato VTABLE, che contiene gli indirizzi di tutte le funzioni virtuali contenute nella struttura.
Nota: con xp->g(); viene generata una chiamata virtuale invece con l'istruzione x.g(); non si genera una chiamata virtuale in quanto il compilatore conosce il tipo di x.
Ogni volta che si eredita un nuovo tipo, il compilatore crea una nuova VTABLE per quel tipo, se non si ridefinisce una funzione, il compilatore usa l'indirizzo di funzione della VTABLE della classe precedente.

Uso del polimorfismo:

I passi necessari per estendere un programma orientato agli oggetti, in C++ sono:

1. Specializzazione delle classi derivate: definita una classe se ne può derivare un'altra definita su un sotto insieme degli oggetti della classe precedente
2. Strutture dati eterogenee: per manipolare un insieme di oggetti simili ma non perfettamente uguali.
3. Gestione di gerarchie di classi
4. Oggetti di classi differenti usati come parametri di una funzione che ha solo funzioni membro comuni a quelle classi.

Funzioni virtuali pure

Una funzione al corpo del quale viene assegnato il valore 0, è chiamata funzione virtuale pura (tali funzioni devono trovarsi all'interno delle classi).
Ogni classe che contiene una o più funzioni virtuali pure è chiamata classe base astratta pura; una classe di questo genere viene usata solo come interfaccia per altre classi derivate da essa.
Le funzioni virtuali pure devono essere ridefinite in una classe derivata prima di crearne un'istanza; se in una classe derivata non si definiscono una o più funzioni virtuali pure, allora tale classe è anch'essa astratta pura.
Una classe astratta è una classe che non ha istanze, e che non è progettata per essere usata per creare oggetti, ma solo per essere ereditata.
Essa specifica una interfaccia a un certo livello di ereditarietà, e fornisce una base per la creazione di altre classi.
Le classi astratte corrispondono a concetti generali, spesso non traducibili direttamente in oggetti specifici, ma utili per fornire una descrizione delle caratteristiche comuni a oggetti abbastanza diversi tra loro.
Le classi astratte dichiarano le funzioni membro che dovranno essere implementate dalle classi derivate, possono essere pensate come lo scheletro del programma, la sua struttura portante attorno alla quale vengono costruite le gerarchie.
Poiché una classe astratta fornisce una struttura per la creazione di altre classi, non occorre creare uno schema di codice e poi copiarlo e modificarlo per creare nuove funzionalità; utilizzando una classe astratta è possibile modificare l'interfaccia e immediatamente propagare le modifiche attraverso il sistema senza errori.
Tutti i cambiamenti fatti dal programmatore che ha derivato la nuova classe sono mostrati esplicitamente nel codice della classe derivata, ed eventuali errori sono quasi sempre isolati in tale codice.

Costruzione e distruzione

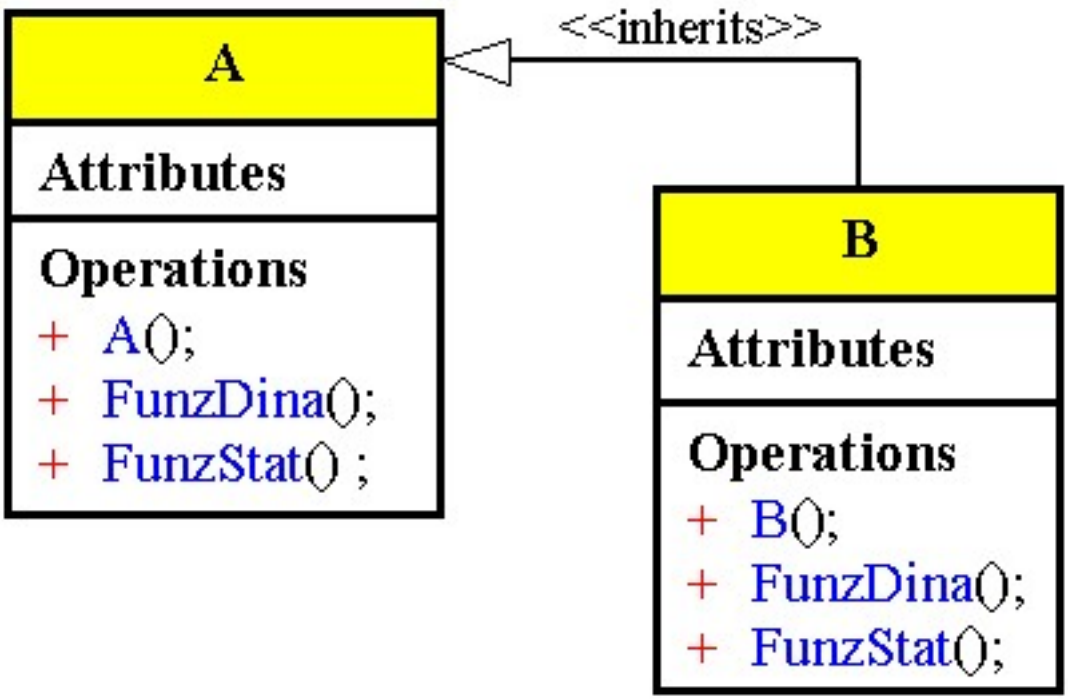
I costruttori e i distruttori sono diversi dalle normali funzioni membro, in quanto essi creano o cancellano oggetti.
Al loro interno il meccanismo delle funzioni virtuali non funziona.
Quando un oggetto viene creato, viene chiamato per primo il costruttore della classe base, poi il costruttore della prossima classe derivata e così via.
L'oggetto è creato dal basso verso l'alto.
Se all'interno della classe base vi fosse una chiamata virtuale essa sarebbe legata a una funzione definita in una classe derivata, ma poiché ci si trova all'interno del costruttore della classe base, gli oggetti della classe derivata non sono ancora stati inizializzati dal proprio costruttore
La stessa logica vale per i distruttori, poiché l'oggetto viene distrutto dall'alto verso il basso, quando si arriva alla classe base, la maggior parte dell'oggetto è già stata distrutta.
Non si possono avere distruttori virtuali puri.

Ritorna

```
/* Binding.cpp
 * esempio autoreferente di binding statico e dinamico
 */
#pragma hdrstop
#include <condefs.h>
#include <iostream.h>
#include "ClassA.h"
//---------------------------------------------------------------------------
USEUNIT("ClassA.cpp");
//---------------------------------------------------------------------------
#pragma argsused
void Attesa(char *);
int main(int argc, char* argv[]) {
  // definizioni
  A * a;
  B * b;
  // inizializzazione
  a = new A ();
  b = new B ();
  cout << "Funzioni dell'oggetto a della classe A" << endl;
  a -> FunzDina ();
  a -> FunzStat ();
  cout << endl;
  cout << "Funzioni dell'oggetto b della classe B" << endl;
  b -> FunzDina ();
  b -> FunzStat ();
  cout << endl;
  // grazie al polimorfismo, la variabile a pu• assumere  come
  // valore un puntatore ad un oggetto della classe B
  a = b;
  cout  << "Funzioni dell'oggetto a di classe A" << endl
        << "a cui e' stato assegnato un valore della classe B" << endl;
  a -> FunzDina ();
  // chiama la funzione non virtuale della classe A
  a -> FunzStat ();
  Attesa("terminare");
  return 0;
}
void Attesa(char * str) {
  cout << "\n\n\tPremere return per " << str;
  cin.get();
}
```

```
/* ClassA.h
 */
//---------------------------------------------------------------------------
#ifndef ClassAH
#define ClassAH
// definisce una classe A con due funzioni che stampano i loro nomi

class A {
public:
  A () { }
  virtual void  FunzDina ();
  void  FunzStat ();
};

// definisce una classe B, derivata da A, che ridefinisce entrambe le funzioni

class B : public A {
public:
  B () { }
  void  FunzDina ();
  void  FunzStat ();
};
//---------------------------------------------------------------------------
#endif


//---------------------------------------------------------------------------
/* ClassA.cpp
 */
//---------------------------------------------------------------------------
#pragma hdrstop
#include <iostream.h>
#include "ClassA.h"
//---------------------------------------------------------------------------
#pragma package(smart_init)
void A::FunzDina () {
  cout << "Funzione dinamica della classe A" << endl;
}
void A::FunzStat () {
  cout << "Funzione statica della classe A" << endl;
}
void B::FunzDina () {
  cout << "Funzione dinamica di classe B" << endl;
}
void B::FunzStat () {
  cout << "Funzione statica di classe B" << endl;
}
```

Diagramma U. M. L. delle classi

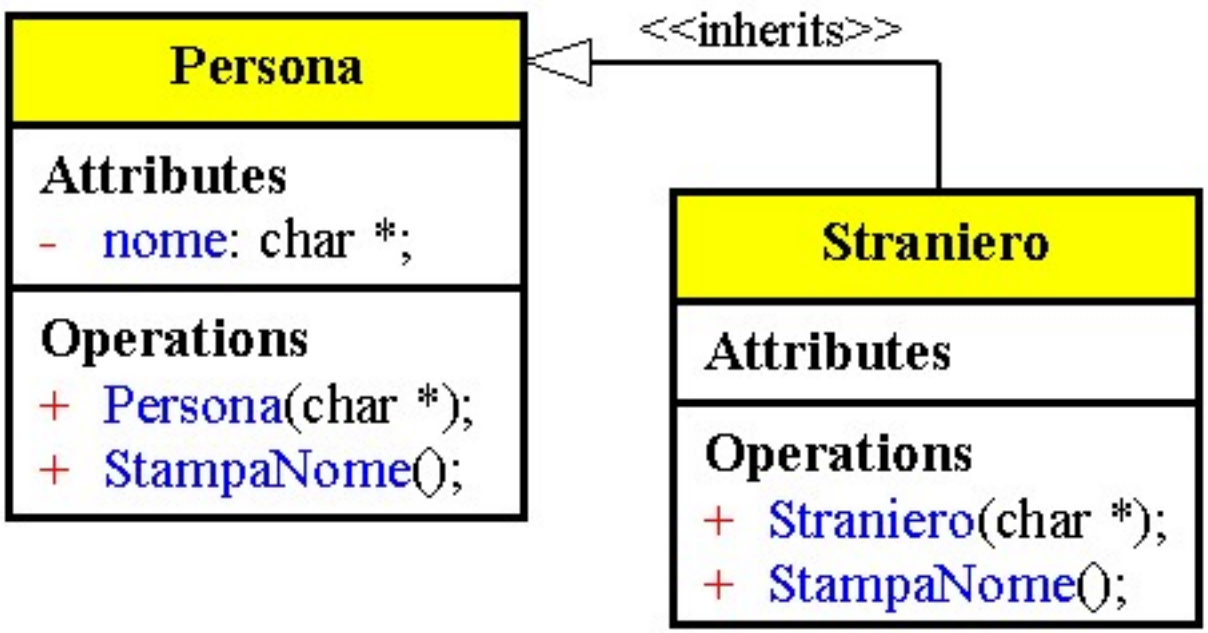


Ritorna

```
/* Foreign.cpp
 * esempio di polimorfismo fra classi derivate
 */
#pragma hdrstop
#include <condefs.h>
#include <iostream.h>
#include "Persona.h"
//---------------------------------------------------------------------------
USEUNIT("Persona.cpp");
//---------------------------------------------------------------------------
#pragma argsused
void Attesa(char *);
int main(int argc, char* argv[]) {
  Persona * uomo;
  Persona * donna;
  // inizializzazione
  uomo = new Persona ("Ernesto");
  donna = new Straniero ("Mary");
  // stampa dei nomi
  cout << "Stampa il nome dell'uomo: " << endl;
  uomo -> StampaNome ();
  cout << "Stampa il nome della donna: " << endl;
  donna -> StampaNome ();
  // e' possibile assegnare alla variabile uomo un puntatore ad
  // un oggetto di classe Straniero, derivata dal Persona
  uomo  = new Straniero ("Ernest");
  cout  << endl;
  cout  << "Ristampa il nome dell'uomo dopo il nuovo assegnamento: "
        << endl;
  uomo  -> StampaNome ();
  Attesa("terminare");
  return 0;
}
void Attesa(char * str) {
  cout << "\n\n\tPremere return per " << str;
  cin.get();
}
```

```
/* Persona.h
 */
//---------------------------------------------------------------------------
#ifndef PersonaH
#define PersonaH
class Persona {
protected:
  char * nome;
public:
  Persona ( char * n ) : nome (n)
  { }
  // funzione virtuale, che usa il binding dinamico
  virtual void  StampaNome ();
};

class Straniero : public Persona {
public:
  Straniero (char * n) : Persona (n)
  { }
  // ridefinisce la funzione virtuale
  void  StampaNome ();
};
//---------------------------------------------------------------------------
#endif

//---------------------------------------------------------------------------
/* Persona.cpp
 */
//---------------------------------------------------------------------------
#pragma hdrstop
#include <iostream.h>
#include "Persona.h"
//---------------------------------------------------------------------------
#pragma package(smart_init)
void Persona::StampaNome () {
  cout << "Il mio nome e' " << nome << endl;
}
void Straniero::StampaNome () {
  cout << "My name is " << nome << endl;
}
```

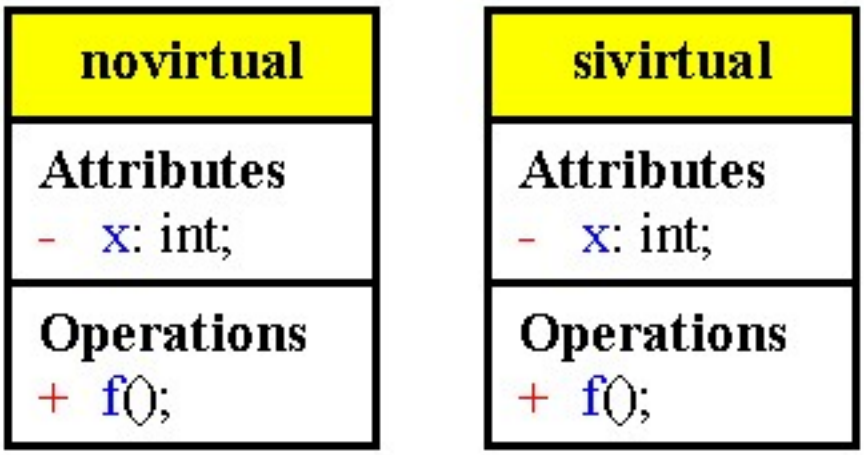
Diagramma U. M. L. delle classi



Ritorna

```cpp
/* Vptrsize.cpp
 * sizeof() scopre l'esistenza di VPTR
 */
#pragma hdrstop
#include <condefs.h>
#include <iostream.h>

//---------------------------------------------------------------------------
#pragma argsused
class novirtual {
public:
  void f() {}
private:
  int x;
};

class sivirtual {
public:
  virtual void f() {}
private:
  int x;
};
void Attesa(char *);
int main(void) {
  cout << "sizeof(novirtual) = " << sizeof(novirtual) << endl;
  cout << "sizeof(sivirtual) = " << sizeof(sivirtual) << endl;
  Attesa("terminare");
  return 0;
}
void Attesa(char * str) {
  cout << "\n\n\tPremere return per " << str;
  cin.get();
}
/*
 * Per effettuare il dynamic binding di una funzione in una classe, si dichiara
 * la funzione con la parola chiave virtual.
 * Quando si dichiara virtuale una funzione di una classe, il compilatore
 * aggiunge alla classe, in maniaera trasparente, un oggetto membro, chiamato
 * VPTR, che e` un puntatore a una tabella di puntatori a funzioni, chiamata
 * VTABLE, che contiene a sua volta i puntatori a tutte le funzioni della
 * classe o delle classi derivate, che sono state dichiarate virtuali.
 */
```

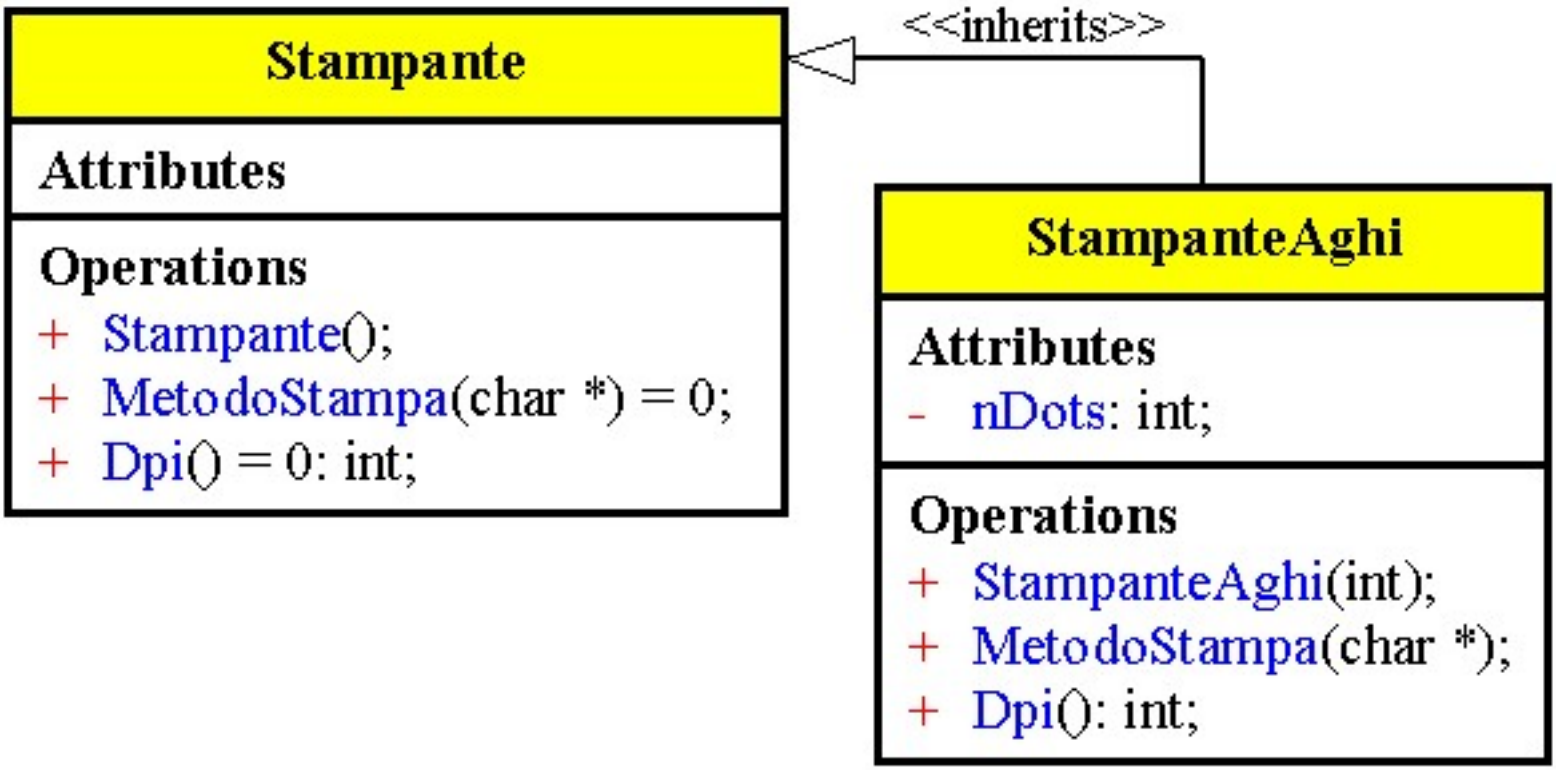
Diagramma U. M. L. delle classi



Ritorna

```
/* Abstr2.cpp
 * esempio di derivazione di una classe "concreta" da una astratta
 */
#pragma hdrstop
#include <condefs.h>
#include <iostream.h>

//---------------------------------------------------------------------------
#pragma argsused
class Stampante {
public:
  Stampante ()
  { }
  virtual void MetodoStampa (char * buffer) = 0;
  virtual int Dpi () = 0;
};

class StampanteAghi : public Stampante {
public:
  StampanteAghi (int n) : nDots (n)
  { }
  void  MetodoStampa (char * buffer)  {
    sprintf (buffer, "Stampante ad aghi con %d aghi", nDots);
  }
  int Dpi () {
    switch (nDots) {
      // valori quasi casuali
      case 8: return 150;
      case 9: return 160;
      case 24: return 300;
      default: return 100;
    }
  }
private:
  int nDots;
} ;
void Attesa(char *);
int main(void) {
  char  descrizione [40];
  StampanteAghi dmp (24);
  dmp.MetodoStampa (descrizione);
  cout << descrizione << endl
       << "DPI: " << dmp.Dpi () << endl
       << endl;
  // definizione di un puntatore a stampanti generiche
  Stampante *   pStamp;
  pStamp = new StampanteAghi (9);
  pStamp -> MetodoStampa (descrizione);
  cout << descrizione << endl
       << "DPI: " << pStamp -> Dpi () << endl
       << endl;
  Attesa("terminare");
  return 0;
}
void Attesa(char * str) {
  cout << "\n\n\tPremere return per " << str;
  cin.get();
}
```

Diagramma U. M. L. delle classi



Ritorna

```
/* Pvdest.cpp
 * Non si possono avere distruttori virtuali puri
 */
#pragma hdrstop
#include <condefs.h>
#include <iostream.h>

//---------------------------------------------------------------------------
#pragma argsused
struct base {
  virtual ~base() = 0;
  // Il compilatore non genera errore, ma il linker si
};

struct derivata : base {
  ~derivata()   {
    cout << "~derivata" << endl;
  }
};
void Attesa(char *);
int main(void) {
  delete new derivata;
  Attesa("terminare");
  return 0;
}
void Attesa(char * str) {
  cout << "\n\n\tPremere return per " << str;
  cin.get();
}
/*
 * Il compilatore accetta questo codice ed effettua la normale sequenza di
 * chiamate dei distruttori, inclusa la chiamata al distruttore della classe
 * base; il linker, tuttavia, non e` in grado di risolvere tale chiamata.
 */
```

Classi contenitore

Una classe contenitore sono classi che contengono altri oggetti
La lista di inizializzazione del costruttore è il luogo nel quale vengono inizializzati tutti gli oggetti membri e gli oggetti della classe base se si usa l'ereditarietà; questo permette di inizializzare tutti gli oggetti membro prima di entrare nel corpo del costruttore.
Le operazioni comunemente applicabili ad un oggetto contenitore sono:

- inserire un nuovo elemento
- rimuovere un elemento esistente
- trovare un elemento
- iterare su ogni elemento del contenitore

Per poter accedere a parti diverse della classe occorre separare il concetto di contenimento dal concetto di controllo e lo si può ottenere per mezzo di un iteratore il quale può agire come un indice di un array in C.
L'iteratore deve avere il permesso di accedere agli elementi privati del contenitore, a questo scopo si utilizza la dichiarazione di tipo friend, inoltre l'iteratore deve conoscere la classe a cui si riferisce e a questo scopo contiene un riferimento a una istanza della classe contenitore.
Dopo che l'iteratore è stato riferito ad una istanza del contenitore tutto il lavoro viene svolto usando l'iteratore e il contenitore diventa un recipiente passivo, in questo modo è possibile creare più iteratori per accedere simultaneamente a più parti del contenitore.

Modelli o template

Si può usare la clausola template per dichiarare un'intera famiglia di classi o di funzioni
Concetto di generalità = abilità di definire moduli parametrici; si tratta di uno schema di modulo, i parametri rappresentano i tipi.
I moduli effettivi chiamate istanze sono ottenuti fornendo tipi effettivi per ogni parametro generico.
Lo scopo della genericità consiste nella possibilità di definire una classe senza specificare il tipo di uno o più dei suoi membri, in questo modo si può modificare la classe per adattarla ai vari usi senza doverla riscrivere.
I modelli permettono di definire una famiglia di classi e possono essere usati per parametrizzare classi o funzioni.
Dichiarazione di un modello

```
template<class T>
```

Definizione di una funzione membro

```
template<class T> tipo_ritorno nome_classe<T>::nome_membro(lista_argomenti) {...}
```

Definizione di un oggetto

```
nome_classe<tipo> identificatore_oggetto(lista_argomenti);
```

Il compilatore crea tutte le definizioni necessarie dovunque venga usato un nuovo tipo e verifica che non vi siano definizioni multiple.
Il processo di creazione di una classe da un modello viene chiamato istanziazione.
Il compilatore genera una copia del codice relativo alle funzioni membro generiche per ogni diversa istanza della classe template.
I template rispetto alle macro forniscono una maggiore sicurezza per il controllo dei tipi e il collegamento del codice.
Inoltre si possono dichiarare parecchie istanze diverse di una classe template in un singolo file sorgente (con la versione macro e quella polimorfa creano problemi)
Se il codice della classe template esegue alcune operazioni sul tipo generico queste devono essere tutte permesse sul tipo effettivo passato come parametro.
Prima di definire un'istanza di un template il compilatore controlla che il tipo effettivo aderisca a tale regola.
Il compilatore esegue l'analisi degli errori solamente quando viene istanziato un modello.

Tipi parametrizzati definiti per mezzo del preprocessore

Il modo più facile per creare un tipo parametrizzato consiste nel definire una normale classe, verificare che essa abbia il comportamento desiderato e poi convertirla in macro:
Regole:

- non si possono inserire commenti all'interno delle macro
- la macro è preprocessata in una singola linea e il compilatore genera un errore per l'intera linea.

Si può emulare la genericità con l'ereditarietà e il polimorfismo; si definisce una classe contenitore capace di gestire i puntatori a una classe di oggetti e la si può adoperare per gestire gli oggetti di qualsiasi classe derivata da quella originale.

Confronto tra template e polimorfismo

Una funzione è polimorfa se uno dei suoi parametri può assumere diversi tipi di dati che devono essere derivati da quello del parametro formale; possono essere eseguite dinamicamente con parametri di tipi diversi.
Una funzione template è tale solo se è preceduta da una specifica clausola template, è soltanto uno schema e non una vera funzione.
Come risultato si ha una famiglia di funzioni sovraccaricate che hanno tutte lo stesso nome e parametri diversi.
Possono essere compilate staticamente in versioni diverse per soddisfare parametri di tipi di dati diversi.

Argomenti dei modelli

Un modello può avere molti argomenti
quando si crea una funzione non inline, tutti gli argomenti devono essere racchiusi tra parentesi angolari, anche se tali argomenti non vengono usati all'interno della funzione.
I parametri di un template possono essere stringhe di caratteri, nomi di funzioni ed espressioni costanti.
Un parametro costante può anche avere un valore di default.
La regola di compatibilità rimane invariata, due istanze sono compatibili se i loro argomenti di tipo sono uguali e le espressioni dei loro argomenti costanti hanno lo stesso valore.

Membri di tipo static nei modelli

Quando si pone un membro di tipo static all'interno di una classe, viene allocato per esso un solo blocco di memoria, condiviso da tutti gli oggetti di quella classe.
I dati membri di tipo static possono essere pensati come un mezzo per comunicare tra gli oggetti di una classe. Ciò vale anche per l'ereditarietà; tutti gli oggetti di una classe e tutti gli oggetti delle sottoclassi condividono lo stesso blocco di memoria per un membro di tipo static.
I modelli si comportano in modo diverso: quando si istanzia un modello, a tutti i membri di tipo static viene assegnato un diverso blocco di memoria

Modelli di funzioni

Un modello di funzione definisce una famiglia di funzioni ognuna delle quali opera con un particolare tipo di dato.
Le funzioni template non devono essere istanziate, i parametri delle chiamate di funzione determinano quale versione della funzione debba essere eseguita.
Invece di effettuare delle conversioni di tipo sui parametri il compilatore genera diverse versioni della funzione anche con tipi compatibili.

Modelli e puntatori a membri

In C++ una struttura contiene oltre ai dati anche funzioni membro, che però non occupano memoria all'interno della struttura.
Le funzioni fanno parte concettualmente della struttura tuttavia fisicamente sono come normali funzioni.
Per un puntatore a un membro della classe il compilatore deve determinare se deve calcolare un offset di un oggetto fisico oppure se deve trovare una funzione.
In entrambi i casi il dereferenziamento del puntatore è legato al puntatore this dell'oggetto.
Un puntatore a un membro di una classe non rappresenta un indirizzo di un elemento fisico ma il luogo nel quale tale elementi si troverà per un particolare oggetto, per questo non è possibile dereferenziare un puntatore a un membro se non è associato a un oggetto.
Quando si definisce un puntatore a un membro di una classe deve essere fornito il tipo esatto del membro referenziato oltre al nome della classe
I puntatori a membri di classi possono essere utili se si vuole ritardare la selezione di una specifica funzione fino a tempo di esecuzione.

Funtore

Un funtore è una classe il cui unico scopo è contenere una funzione.
La sola ragione per creare un oggetto di tale classe è di chiamare la sua funzione.
Un funtore ha lo stesso ruolo di una funzione ma può essere creato, passato come parametro e trattato come oggetto.
Se si usa l'ereditarietà con un funtore si può ridurre la duplicazione del codice inserendo codice comune nella classe base.

Modelli e ereditarietà

Esempio: Si vuole creare una classe di stack ma di lasciare all'utente la possibilità di scegliere se implementare lo stack con un array di dimensione fissa, con un array allocato sullo heap o con una lista collegata

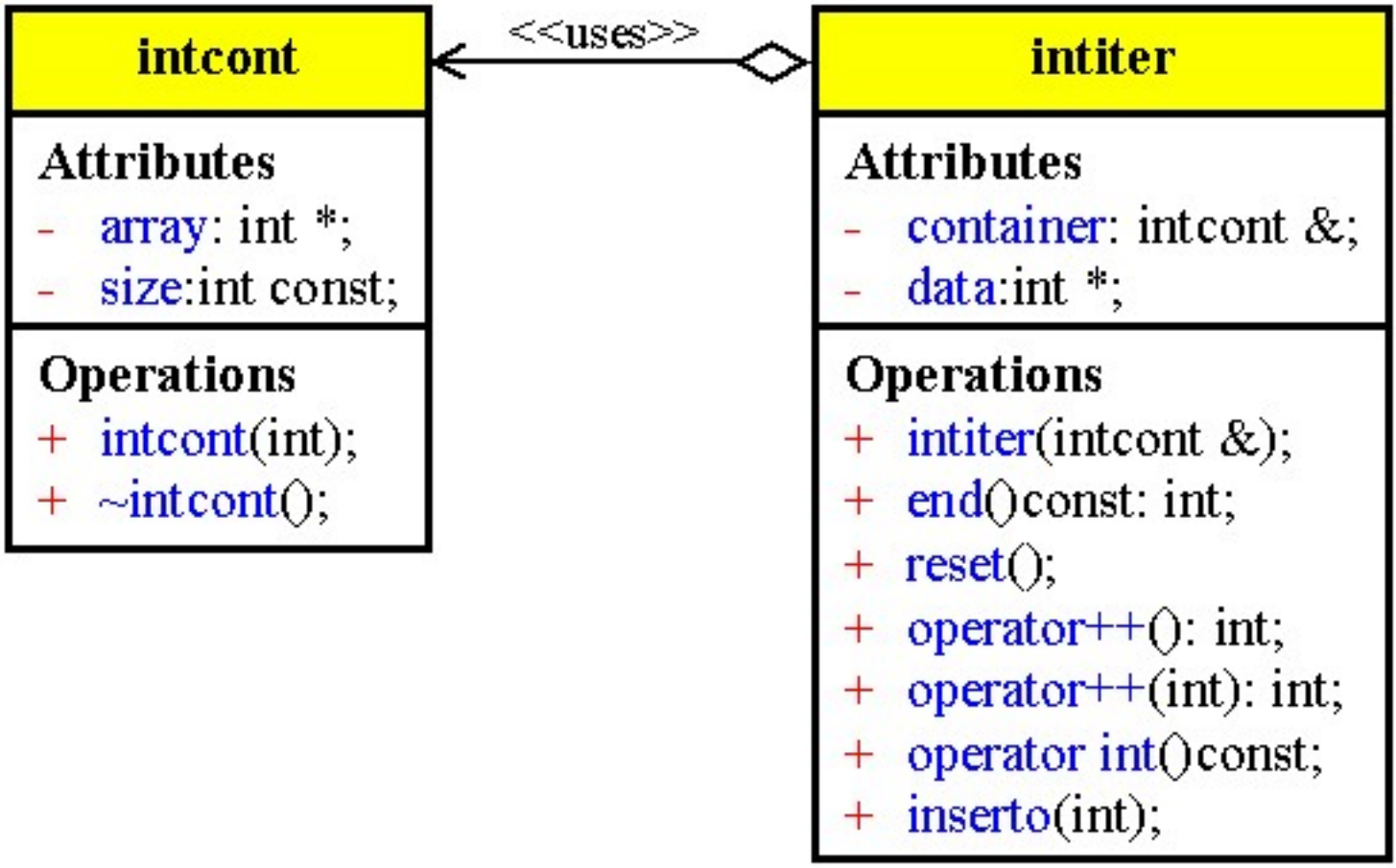
1. L'array di dimensione fissa è più efficiente, sia per quanto riguarda l'inizializzazione, che il tempo di accesso; esso quindi rappresenta la soluzione migliore per oggetti che devono essere creati rapidamente, in gran numero, specialmente se sono passati per valore.
2. L'array allocato sullo heap ha un tempo di accesso veloce, ma l'inizializzazione è più lenta, poiché deve essere chiamato l'operatore new. Tuttavia, si può determinare la dimensione dello stack a tempo di esecuzione, anche se, dopo la creazione resta fissa.
3. La lista collegata rappresenta la soluzione più flessibile, poiché crea spazio sullo heap per ciascun elemento che si aggiunge allo stach, e quindi non si esaurirà mai la memoria dello stack; è anche il metodo più lento , e passare la lista per valore diventa molto complesso.

Ritorna

```
/* Pintiter.cpp
 */
#pragma hdrstop
#include <condefs.h>
#include <iostream.h>
#include "Intiter.h"
//---------------------------------------------------------------------------
USEUNIT("Intiter.cpp");
//---------------------------------------------------------------------------
#pragma argsused
void Attesa(char *);
int main(int argc, char* argv[]) {
  intcont X(15);
  intiter I(X);
  for(int i = 0; !I.end(); I++, i++)
    I.inserto(1 << i);
  I.reset();
  while(!I.end()) {
    cout << I << endl;
    I++;
  }
  Attesa("terminare");
  return 0;
}
void Attesa(char * str) {
  cout << "\n\n\tPremere return per " << str;
  cin.get();
}
/*
 * L'iteratore deve avere il permesso di accedere agli elementi privati del
 * contenitore; a questo scopo si utilizza la dichiarazione di tipo friend
 * alla fine della classe intcont.
 * L'iteratore deve conoscere la classe a cui si riferisce, e a questo scopo
 * contiene un riferimento a un'istanza della classe intcont chiamata
 * container
 */
```

```
/* Intiter.h
 * Classe contenitore di interi con iteratore
 */
//---------------------------------------------------------------------------
#ifndef IntiterH
#define IntiterH
#include <string.h>    // memset()
class intcont {
public:
  intcont(int);
  ~intcont() { delete []array; }
  friend class intiter;
private:
  int *array;
  const size;
};

class intiter {
public:
  intiter(intcont &ic) : container(ic), data(ic.array) {}
  int end() const {
    // E` alla fine?
    return (data - container.array) >= container.size;
  }
  void reset() {
    data = container.array;                    // Inizio lista
  }
  int operator++();
  int operator++(int) { return operator++(); }    // Postfisso
  operator int() const { return *data; }     // Conversione di tipo
  void inserto(int X) { *data = X; }
private:
  intcont &container;
  int *data;
};
//---------------------------------------------------------------------------
#endif

//---------------------------------------------------------------------------
/* Intiter.cpp
 * Classe contenitore di interi con iteratore
 */
//---------------------------------------------------------------------------
#pragma hdrstop

#include "Intiter.h"
//---------------------------------------------------------------------------
#pragma package(smart_init)
intcont::intcont(int sz) : size(sz) {
  array = new int[sz];
  memset(array, 0, size * sizeof(int));
}
int intiter::operator++() {
  // Operatore prefisso
  if(!end()) {
    data++;
    return 1;
  }
  return 0;             // Indica che la lista e` finita.
}
```

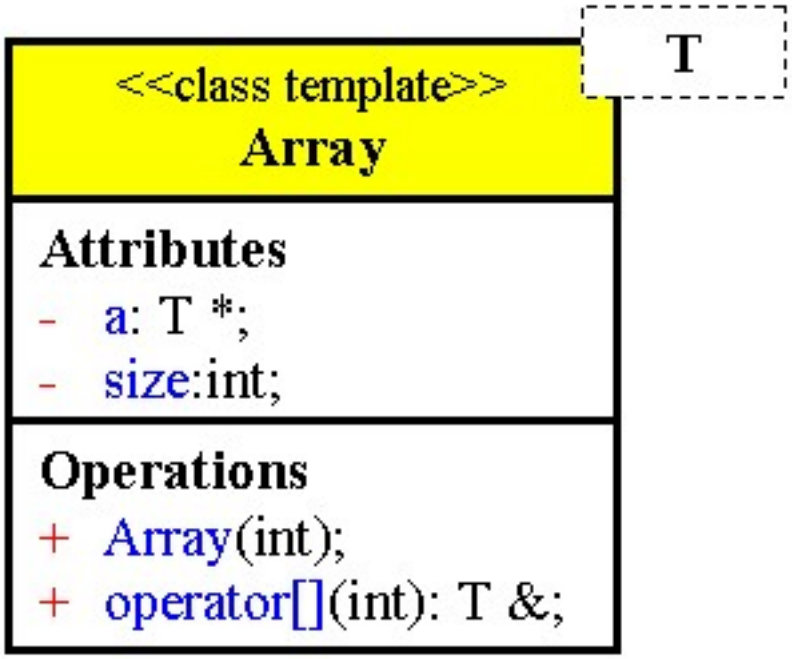
Diagramma U. M. L. delle classi



Ritorna

```
/* Simtempl.cpp
 * Uso molto semplice di modelli
 */
#pragma hdrstop
#include <condefs.h>
#include <iostream.h>

//---------------------------------------------------------------------------
#pragma argsused
template<class T>
class Array {
public:
  Array(int);
  T &operator[](int index);
private:
  T *a;
  int size;
};

template<class T>
Array<T>::Array(int sz) : a(new T[size = sz]) {
  for(int i = 0; i < size; i++)
    a[i] = 0;
}

template<class T>
T &Array<T>::operator[](int index) {
  if(index >= 0 && index < size)
    return a[index];
  clog << "Array::operator[] indice fuori limite\n";
  return *new T;
  // Assume un costruttore di default
}
void Attesa(char *);
int main(int argc, char* argv[]) {
  Array<int>I(10);                    // Istanziazione di un modello
  for(int i = 1; i < 11; i++) {
    I[i] = (2 << i);
    cout << I[i] << endl;
  }
  Attesa("terminare");
  return 0;
}
void Attesa(char * str) {
  cout << "\n\n\tPremere return per " << str;
  cin.get();
}
/*
 * Il modello per la classe Array puo` contenere qualunque tipo.
 * Il costruttore alloca un determinato numero di puntatori al tipo T e li
 * inizializza a 0.
 * La funzione overloaded operator[] ritorna un riferimento a un tipo
 * arbitrario, in modo da poter leggere e scrivere nell'array, e controlla
 * che non vengano superati i limiti dell'array; nel caso cio` avvenga, viene
 * memorizzato un messaggio di errore nell'oggetto di tipo iostream clog ( che
 * in seguito, stampa il messaggio)
 */
```

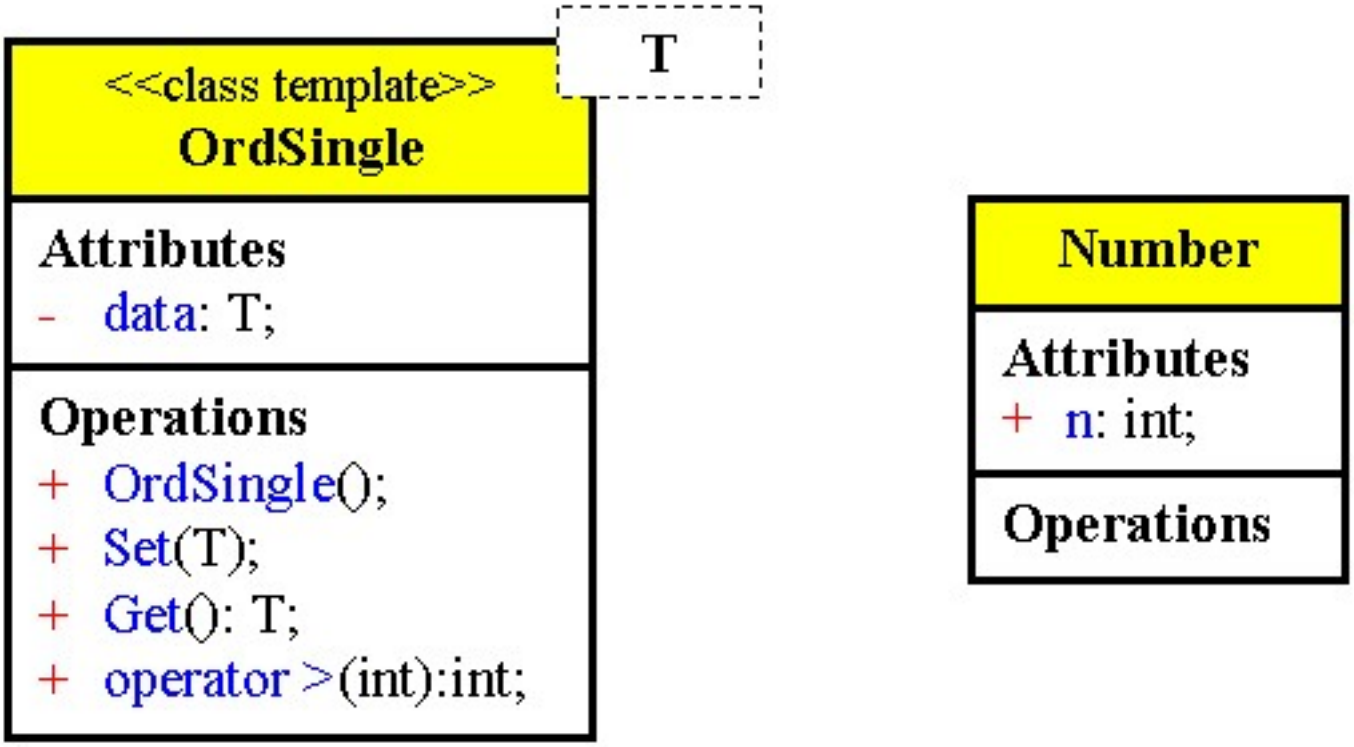
Diagramma U. M. L. delle classi



Ritorna

```
/* Typerr.cpp
 * esempio di come un errore nel parametro di tipo produca
 * un messaggio d'errore fuorviante
 */
#pragma hdrstop
#include <condefs.h>
#include <iostream.h>
//---------------------------------------------------------------------------
#pragma argsused
// definisce una classe simile a quella dell'esempio precedente,
// con una relazione d'ordine
template <class T>
class OrdSingle {
public:
  OrdSingle() { }
  void Set (T el) {
    data = el;
  }
  T Get() {
    return data;
  }
  int operator > (int value) {
    return (data > value);
  }
private:
  T data;
};

// definisce una classe numerica
class Number {
public:
  int   n;
  //    int operator >(int i) { return (n > i); }
};
void Attesa(char *);
int main(int argc, char* argv[]) {
  // istanzia una coppia di oggetti
  OrdSingle <int> os1;
  OrdSingle <Number> os2;
  Attesa("terminare");
  return 0;
}
void Attesa(char * str) {
  cout << "\n\n\tPremere return per " << str;
  cin.get();
}
```

Diagramma U. M. L. delle classi

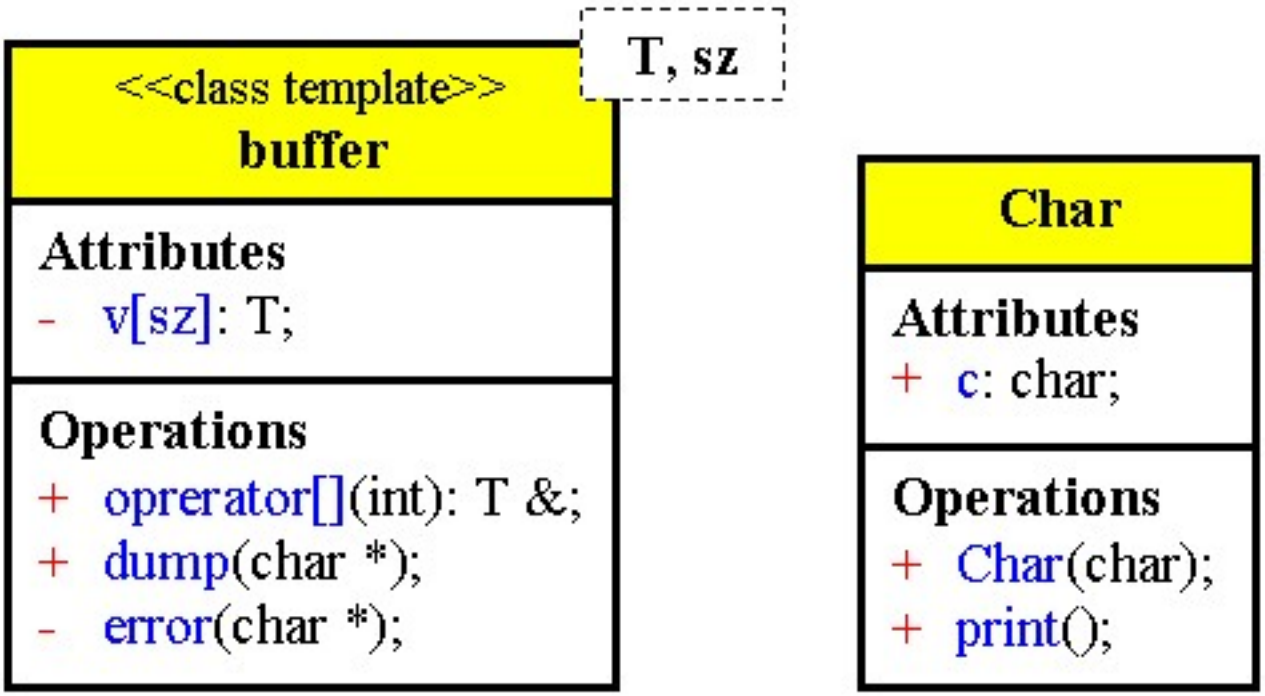


Ritorna

```
/* Args.cpp
 * Gli argomenti dei modelli possono anche essere tipi predifiniti
 * In questo caso sz e` trattata come una vera costante all'interno della classe
 */
#pragma hdrstop
#include <condefs.h>
#include <iostream.h>
#include "Buffer.h"
//---------------------------------------------------------------------------
#pragma argsused
void Attesa(char *);
int main(int argc, char* argv[]) {
  buffer<Char, 20> buf1;
  buffer<Char, 20> buf3;
  buffer<Char, 100> buf4;
  // buf = buf4  // Errore
  buf1[0] = 'h';                                  // operator[] overloaded
  buf1[1] = 'e';
  buf1[2] = 'l';
  buf1[3] = 'l';
  buf1[4] = 'o';
  buf3 = buf1;                            // operator= generato automaticamente
  buf3.dump();
  buf1[21] = 'x';                         // Fuori limiti
  Attesa("terminare");
  return 0;
}
void Attesa(char * str) {
  cout << "\n\n\tPremere return per " << str;
  cin.get();
}
/*
 * Quando si crea una funzione non inline, tutti gli argomenti devono essere
 * racchiusi tra parentesi angolari, anche se tali argomenti non vengono usati
 * all'interno della funzione.
 * L'argomento del modello sz, viene sostituito dal compilatore all'interno
 * del modello, usando il valore attuale.
 * Cio`significa che e` possibile determinare la dimensione di un oggetto a
 * tempo di compilazione, usando un argomento del modello, cio` ha un limite:
 * non e` piu` possibile creare oggetti di dimensione differente a tempo di
 * esecuzione.
 */
```

```
/* Buffer.h
 */
#ifndef BufferH
#define BufferH
#include <iostream.h>
#include <stdlib.h>

template<class T, int sz>
class buffer {
public:
  T &operator[](int index) {
    if(index < 0 || index >= sz)
      error("Fuori limiti");
    return v[index];
  }
  void dump(char *msg = "");
private:
  T v[sz];              // Inizializzazione automatica dell'array
  void error(char *msg = "") {
    clog << endl << "Errore nel buffer: " << msg << endl;
    Attesa("terminare");     // Come fa a conoscere Attesa() ????
    exit(1);
  }
};
// Definizione di un modello di funzione all'esterno della classe
template<class T, int sz>
void buffer<T, sz>::dump(char *msg) {
  if(*msg)
    cout << msg;
  for(int i = 0; i < sz; i++)
    v[i].print();                       // Non sa se print() esiste
}

struct Char {
  char c;
  Char(char C = 0) : c(C) {}
  void print() { cout << c; }
};
#endif
```

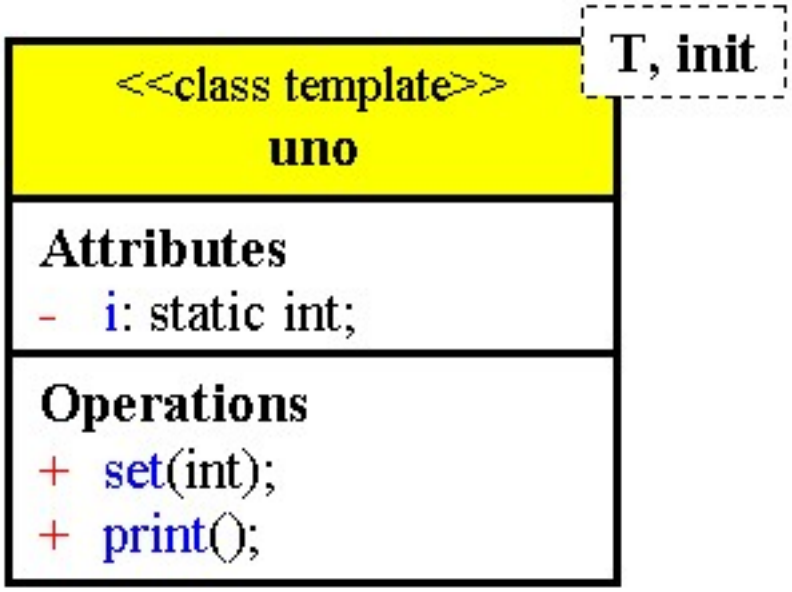
Diagramma U. M. L. delle classi



Ritorna

```
/* Statict.cpp
 * Modelli e membri di tipo static
 */
#pragma hdrstop
#include <condefs.h>
#include <iostream.h>

//---------------------------------------------------------------------------
#pragma argsused
template<class T, int init>
class uno {
public:
  void set(int x) { i = x; }
  void print() {
    cout << "i = " << i << endl;
  }
private:
  static int i;
};
template<class T, int init> int uno<T, init>::i = init;
void Attesa(char *);
int main(int argc, char* argv[]) {
  uno<int, 1> I;
  uno<int, 1> J;
  uno<char, 2> U, V;
  J.print();
  V.print();
  I.set(100);
  U.set(47);
  J.print();
  V.print();
  Attesa("terminare");
  return 0;
}
void Attesa(char * str) {
  cout << "\n\n\tPremere return per " << str;
  cin.get();
}
```

Diagramma U. M. L. delle classi



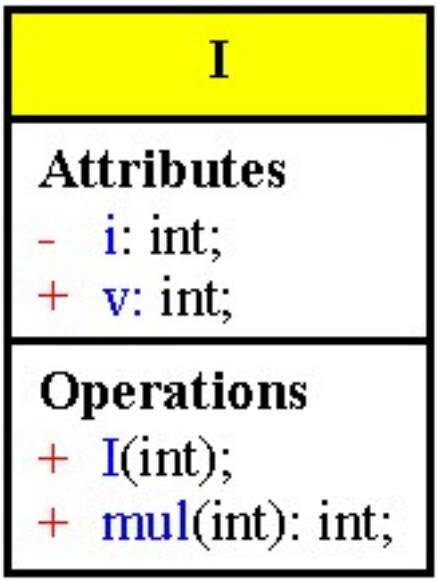
Ritorna

```
/* Tempf1.cpp
 * esempio di dichiarazione ed uso di funzione template
 */
#pragma hdrstop
#include <condefs.h>
#include <iostream.h>

//---------------------------------------------------------------------------
#pragma argsused
template <class T> T Min (T a, T b) {
  if (a < b)
    return a;
  else
    return b;
}
void Attesa(char *);
int main(int argc, char* argv[]) {
  char  c1 = 'W', c2 = 'h';
  int   n1 = 23, n2 = 67;
  cout << Min (c1, c2) << endl;
  cout << Min (n1, n2) << endl;
  Attesa("terminare");
  return 0;
}
void Attesa(char * str) {
  cout << "\n\n\tPremere return per " << str;
  cin.get();
}
```

Ritorna

```
/* Pmemb1.cpp
 * Puntatori a membri di classi
 */
#pragma hdrstop
#include <condefs.h>
#include <iostream.h>

//---------------------------------------------------------------------------
#pragma argsused
class I {
public:
  int v;
  I(int ii = 0): i(ii), v(ii) {}
  int mul(int x) { return i * x; }
private:
  int i;
};
void Attesa(char *);
int main(void) {
  // Definizione e inizializzazione di un puntatore a membro di tipo int
  int I:: *ptr_mem_int = &I::v;
  // Definizione e inizializzazione di un puntatore a membro di tipo function
  int(I:: *ptr_mem_func)(int) = &I::mul;
  I a(4);
  // Il deferenziamento di un puntatore si ottiene facendo precedere
  // l'asterisco dal nome della classe e dall'operatore ::
  a.*ptr_mem_int = 12;
  cout << "a.v = " << a.v << endl;
  int b = (a.*ptr_mem_func)(47);
  cout << "b = " << b << endl;
  // Nel caso di un puntatore ad un oggetto si deve aggiungere all'operatore
  // -> l'asterisco
  I *c = &a;
  c->*ptr_mem_int = 44;
  cout << "a.v " << a.v << endl;
  int d = (c->*ptr_mem_func)(b);
  cout << "d = " << d << endl;
  Attesa("terminare");
  return 0;
}
void Attesa(char * str) {
  cout << "\n\n\tPremere return per " << str;
  cin.get();
}
/*
 * Un puntatore ad un membro di una classe non rappresenta un indirizzo di un
 * elemento fisico, ma il luogo nel quale tale elemento si trovera` per un
 * particolare oggetto.
 * Non e` possibile derefernziare un puntatore a un membro se non e` associato
 * a un oggetto.
 */
```

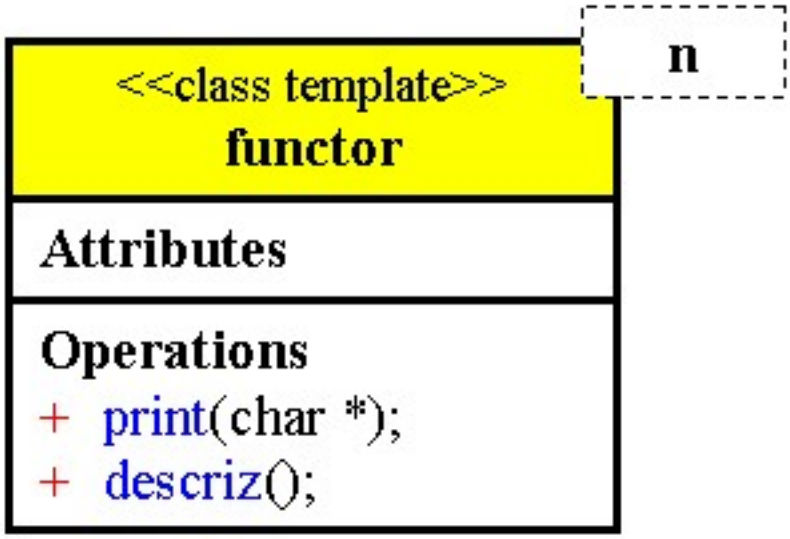
Diagramma U. M. L. delle classi



Ritorna

```
/* Tfunctor.cpp
 * Generazione di un funtore con i modelli
 */
#pragma hdrstop
#include <condefs.h>
#include <iostream.h>

//---------------------------------------------------------------------------
#pragma argsused
template<int n>
class functor {
public:
  void print(char *msg = "") {
    cout << "Funtore " << msg << endl;
  }
  void descriz();               // Definito altrove
};
template<int n> void functor<n>::descriz() {
  cout << "Questo e` il funtore " << n << endl;
}

void Attesa(char *);
int main(void) {
  functor<1>A;
  functor<2>B;
  functor<3>C;
  A.print("A");
  A.descriz();
  B.print("B");
  B.descriz();
  C.print("C");
  C.descriz();
  Attesa("terminare");
  return 0;
}
void Attesa(char * str) {
  cout << "\n\n\tPremere return per " << str;
  cin.get();
}
/*
 * Un funtore e` una classe il cui unico scopo e` contenere una funzione; cioe'
 * la sola ragione per creare un oggetto di tale classe e` di chiamare la sua
 * funzione, ma puo` essere creato, passato come parametro e trattato come un
 * oggetto.
 */
```

Diagramma U. M. L. delle classi



Gestione delle eccezioni

La gestione delle eccezioni consentono di tenere conto del possibile comportamento errato di un'applicazione.
Con il meccanismo della gestione delle eccezioni, una funzione può segnalare un problema al codice che l'ha chiamata e questo può gestire il problema o lasciare che il messaggio raggiunga il livello superiore e così via.
Ogni blocco viene terminato correttamente pulendo lo stack e chiamando i distruttori degli oggetti che terminano il loro intervallo di visibilità.
Eventualmente se nessuno gestisce il problema il programma termina.
Parole chiave:

tray	delimita un blocco di codice che può provocare delle eccezioni
throw	provoca l'eccezione.
catch	gestisce l'eccezione del blocco try precedente.

Esempio: Una funzione richiede che uno dei suoi parametri non sia zero per evitare un errore, e controlla questa condizione.
Se la condizione è verificata genera (throw) un'eccezione, evitando perciò di eseguire il resto del suo codice.
Nel main la chiamata a questa funzione è all'interno di un blocco try: se viene generata un'eccezione durante l'esecuzione di una delle funzioni chiamate all'interno del blocco, nelle istruzioni catch alla fine viene cercato un tipo di parametro corrispondente a quello usato nell'istruzione throw.
Se viene trovata una corrispondenza il controllo passa al blocco che segue il catch corrispondente.
Si lancia (throw) un oggetto e si prende (catch) un tipo.
L'espressione throw richiede:

- un oggetto di un tipo definito dall'utente,
- un tipo aritmetico predefinito,
- una stringa,
- un messaggio

L'istruzione catch corrispondente richiede:

- una classe definita dall'utente,
- un parametro di tipo aritmetico,
- un const char * ecc.

throw

Dopo la definizione di una classe per le eccezioni

```
class EccezClass {.....};
```

Si può lanciare un'eccezione:

```
void f() {
        EccezClass OggettoEcc;
        if(...) throw OggettoEcc;       //Si utilizza una copia dell'oggetto
}
void g() {
        if(...) throw EccezClass();             //Si utilizza un oggetto temporaneo
}
void h() {
        if(...) throw new EccezClass; //Si utilizza un puntatore
}
```

Se viene trovato un gestore in uno dei blocchi o delle funzioni attualmente sullo stack viene chiamato altrimenti il controllo è passato alla funzione terminate() che per default termina il programma con un messaggio.

L'istruzione throw deve sempre avere un parametro per indicare a che oggetto si riferisce l'eccezione, altrimenti viene rilanciata l'eccezione.
Se throw non ha un parametro rilancia l'eccezione

```
catch(EccezClass) {
        if(...) throw;    // Passa l'eccezione al prossimo gestore
}
```

Per essere rilanciata un'eccezione deve essere già stata sollevata altrimenti viene chiamata la funzione terminate..

```
void m() throw(EccezClass) {...}      // Specifica quale eccezione può essere lanciata dalla funzione.
```

Indica che solo il tipo di eccezione EccezClass può essere propagato al di fuori di m(), la funzione m() o altre chiamate da m() possono provocare qualsiasi altro tipo di eccezioni che devono essere gestite tutte all'interno di m()
Una funzione può specificare zero, uno o più tipi di eccezioni

```
void g() throw() {..}                 // Non può provocare alcuna eccezione
void f() throw(EccezClass) {...}      // Si aspetta solo oggetti EccezClass
void h() throw(Errore, FineMemoria, EccezClass*) {...}
                                      // Si aspetta solo oggetti delle classi specificati
void j() {...}                        // Non ha limiti
```

Se un'eccezione di tipo differente da quelle dichiarate viene generata e non gestita all'interno della funzione, il programma chiama la funzione unexpected()

Try - catch

L'istruzione catch può solo seguire un blocco try e può contenere un parametro o l'ellissi.

```
catch(EccezClass eccOgg) {
        // Gestore per l'eccezione di tipo EccezClass
}
catch(...) {
        // Gestore per qualsiasi tipo di eccezione
}
```

Se il blocco try è seguito da più di un'eccezione, queste sono valutate nell'ordine in cui sono scritte; se due istruzioni catch corrispondono all'oggetto lanciato solo la prima sarà eseguita (gerarchia di classi)

Gestione e correzione degli errori

Le eccezioni possono essere usate per gestire gli errori cercando di correggere la situazione
Esempio: In una funzioni dividi() che può funzionare solo con valori positivi, quando un valore è negativo si può cambiarne il segno e riprovare.
Nota: Le eccezioni hanno un ruolo specifico nella programmazione, e non è corretto usarle come istruzioni generiche per il controllo del flusso o per controllare l'input.
Esse dovrebbero essere limitate alla gestione di situazioni eccezionali e rare dovute a fattori esterni o errori.

Srotolamento dello stack

Stack unwinding
Quando viene sollevata un'eccezione, questa può essere immediatamente intercettata da un'istruzione catch o passata a una funzione chiamante.
Nel secondo caso se l'istruzione throw fosse tradotta in un salto diretto al gestore corrispondente, lo stack rimarrebbe sporco e alcune risorse potrebbero essere perse.
L'uso della gestione delle eccezioni necessita di richiedere e rilasciare sempre le risorse nei costruttori e nei distruttori delle classi.
L'uso delle eccezioni richiede un uso appropriato di costruttori e distruttori per evitare errori di memoria e perdita di risorse.

Abortire un costruttore

Come intervenire quando un costruttore non può creare correttamente un oggetto?
I costruttori non possono ritornare direttamente un codice di errore poiché non hanno un valore di ritorno.
Se un costruttore incontra delle condizioni anormali può generare un'eccezione e interrompere la creazione dell'oggetto
Le operazioni eseguite dal costruttore prima dell'istruzione throw sono rischiose, poiché il loro effetto deve essere annullato manualmente.

```
A::A() {
        // Alloca qualcosa
        if(...) {
                // Rilascia quello che è già stato allocato
                throw(...);
        }
}
```

Questo problema coinvolge solo il costruttore della classe che sta effettivamente generando l'eccezione; se questa classe deriva da una classe base, la classe base viene distrutto correttamente.
Se si genera un'eccezione in un costruttore bisogna preoccuparsi di annullare solamente gli effetti del costruttore stesso.
L'uso delle classi per specificare il tipo di eccezione è molto elegante e molto potente

Allocazione della memoria con new

La commissione ANSI C++ ha deciso di aggiungere il supporto per le eccezioni all'operatore new, quando non vi è memoria disponibile new genera un'eccezione di tipo xalloc
Questo comportamento non è compatibile con il vecchio codice; per evitare di dover modificare il vecchio codice si può scegliere di usare il comportamento precedente dell'operatore new disabilitando la funzione new_handler con set_new_handler(0);
La stessa funzione può essere usata per impostare un handler specifico fornito dall'utente.

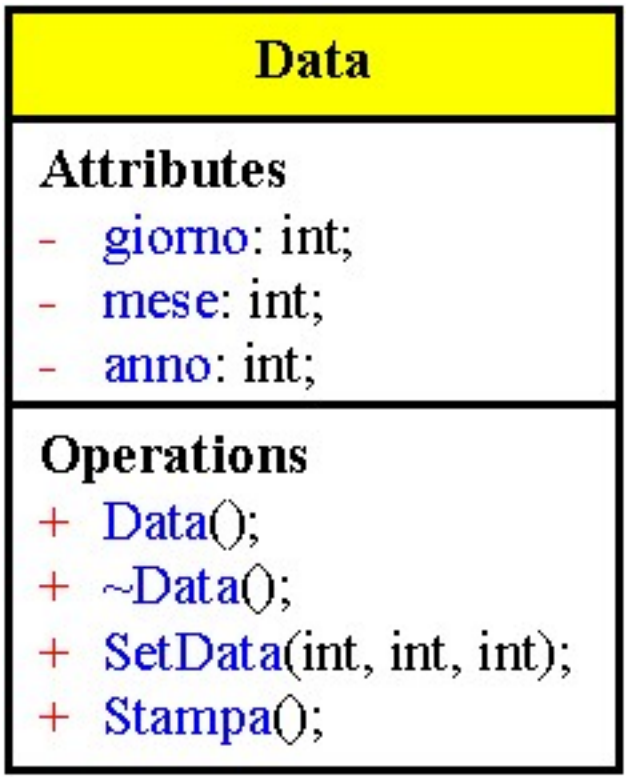
Ritorna

```
/* Basic1.cpp
 * esempio sulle basi della gestione delle eccezioni
 */
#pragma hdrstop
#include <condefs.h>
#include <iostream.h>

//---------------------------------------------------------------------------
#pragma argsused
// dichiara una funzione che richiede due valori interi
// il secondo deve essere diverso da zero
int dividi(int a, int b) {
  if (b == 0)
    throw("Divisione per zero");
  return (a/b);
}
void Attesa(char *);
int main(void) {
  int a, b;
  cout << "Introdurre due valori da dividere: ";
  cin >> a >> b;
  try {
    cout << a << " / " << b << " = " << dividi(a, b) << endl;
  }
  catch (const char *testo)     {
    cout << "ERRORE: " << testo << endl;
  }
  Attesa("terminare");
  return 0;
}
void Attesa(char * str) {
  cout << "\n\n\tPremere return per " << str;
  cin.ignore(4, '\n');
  cin.get();
}
```

Ritorna

```
/* Srotola1.cpp
 * un esempio di srotolamento dello stack
 */
#pragma hdrstop
#include <condefs.h>
#include <iostream.h>

//---------------------------------------------------------------------------
#pragma argsused
class Data {
public:
  Data()  {
    cout << "Data di default creata" << endl;
  }
  ~Data() {
    cout << "Chiamata al distruttore" << endl;
  }
  void SetData(int g, int m, int a) {
    if (g > 31)
      throw("Giorno troppo grande");
    if (m > 12)
      throw("Mese troppo grande");
    giorno = g;
    mese = m;
    anno = a;
  }
  void Stampa() {
    cout << giorno << '-' << mese << '-' << anno << endl;
  }
private:
  int giorno;
  int mese;
  int anno;
};
void Attesa(char *);
int main(void) {
  try {
    Data d1, d2;
    d1.SetData(11, 11, 1993);
    d1.Stampa();
    d2.SetData(11, 15, 1993);
    d2.Stampa();
  }
  catch(const char* descrizione) {
    cout << "Eccezione: " << descrizione << endl;
  }
  Attesa("terminare");
  return 0;
}
void Attesa(char * str) {
  cout << "\n\n\tPremere return per " << str;
  cin.get();
}
```

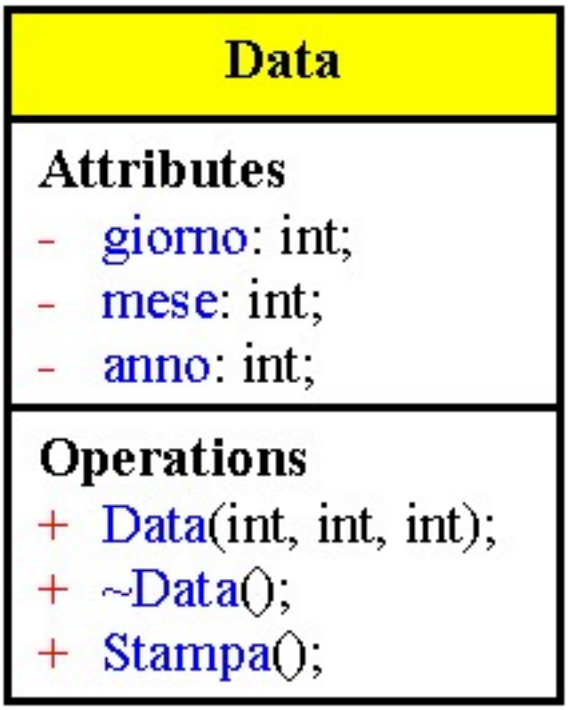
Diagramma U. M. L. delle classi



Ritorna

```
/* Costrut1.cpp
 * esempio di costruttore che genera un'eccezione
 */
#pragma hdrstop
#include <condefs.h>
#include <iostream.h>

//---------------------------------------------------------------------------
#pragma argsused
class Data {
public:
  Data(int g, int m, int a) {
    cout << "Inizio del costruttore... " << endl;
    if (g > 31)
      throw("Giorno troppo grande");
    if (m > 12)
      throw("Mese troppo grande");
    giorno = g;
    mese = m;
    anno = a;
    cout << "Oggetto Data costruito correttamente" << endl;
  }
  ~Data() {
    cout << "Chiamata al distruttore" << endl;
  }
  void Stampa() {
    cout << giorno << '-' << mese << '-' << anno << endl;
  }
private:
  int giorno;
  int mese;
  int anno;
};
void Attesa(char *);
int main(void) {
  try {
    Data d1(11, 11, 1993);
    d1.Stampa();
    cout << endl;
    Data d2(11, 15, 1993);
    d2.Stampa();
  }
  catch(const char* descrizione) {
    cout << "Eccezione: " << descrizione << endl;
  }
  Attesa("terminare");
  return 0;
}
void Attesa(char * str) {
  cout << "\n\n\tPremere return per " << str;
  cin.get();
}
```

Diagramma U. M. L. delle classi



Ritorna

```
/* ImplEcc.cpp
 * Eccezzioni per l'input
 */
#pragma hdrstop
#include <condefs.h>
#include <iostream.h>

//---------------------------------------------------------------------------
#pragma argsused
void Attesa(char *);
int main(int argc, char* argv[]) {
int x;
try {
  cin.exceptions(ios_base::badbit | ios_base::failbit);    //1
  cin >> x;
  // do lots of other stream i/o
}
catch(ios_base::failure& exc)                                  //2
{   cerr << exc.what() << endl;
    throw;
}


  Attesa("terminare");
  return 0;
}
void Attesa(char * str) {
  cout << "\n\n\tPremere return per " << str;
  cin.ignore(4, '\n');
  cin.get();
}
```

Informazioni di typo run-time

Al linguaggio C++ sono stati aggiunti dei controlli sui tipi a tempo di esecuzione RTTI Run Time Type Identification che costituiscono un comportamento dinamico del linguaggio.
Usando l'operatore typeid si può richiedere il tipo di un oggetto o di una espressione.
L'operatore typeid ritorna un oggetto della classe standard Type_info (classe standard definita dal comitato ANSI)

```
class typeinfo {
public:
        virtual ~typeinfo();
        int operator==(const typeinfo &rhs) const;
        int operator!=(const typeinfo &rhs) const;
        int before(const typeinfo &rhs);
        const char *name() const;
private:
        const char *name;
        typeinfo(const typeinfo &rsh);                  // costruttore di copia
        tipeinfo &operator=(const typeinfo &rsh);   // operatore di assegnamento
}
```

L'operatore typeid può essere usato per confrontare due tipi o per estrarre il nome di un tipo.
Esempio:

```
        int x;
        long y;
        if(typeid(y) == typeid(x))
                cout << "Stesso tipo" << endl;
        else
                cout << "Tipo diverso" << endl;
```

Si può verificare se un puntatore si riferisce a una classe particolare usando l'operatore == tra gli oggetti Type_info ritornati dall'operatore typeid
La funzione membro typeinfo::name() restituisce una rappresentazione del tipo in formato *char
Esempio:

```
        double reale;
        cout << typeid.name(reale) << endl;             //Scrive "double"
```

La funzione membro typeinfo::before() permette di sapere se un tipo appartiene a una classe base di un altro tipo
Esempio:

```
        class Number{};
        class Integer: public Number {};
        Number *pnum = new Number;
        Integer *pin = new Integer;
        if(typeid(*pin).before(typeid(*pnum)))
                cout << "*pin precede *pnum" << endl;
        else
                cout << "*pin non precede *pnum"<< endl;
```

typeinfo::before() può essere usata con qualsiasi tipo di oggetto
Quando si usano classi template, si può scrivere codice flessibile e riusabile, ma si perde parte del controllo sui contenuti della classe.
Se c'è bisogno di informazioni run-time per sapere che tipo di template si sta usando è opportuno usare RTTI

RTTI è particolarmente utile con il polimorfismo, quando i tipi coinvolti appartengono a classi virtuali

Esempio:

```
class Number{
public:
        virtual Number() {}
};
class Integer: public Number {};
Number *pnum = new Integer;
Number &ref = *pnum;
cout << typeid.name(pnum);          // Number *
cout << typeid.name(*pnum);         // Integer
cout << typeid.name(ref);           // Integer
```

RTTI non funziona se applicato a classi non virtuali

Esempio:

```
class Number{
public:
        Number() {}
};
class Integer: public Number {};
Number *pnum = new Integer;
Number &ref = *pnum;
cout << typeid.name(pnum);          // Number *
cout << typeid.name(*pnum);         // Number
cout << typeid.name(ref);           // Number
```

RTTI non deve essere inteso come una sostituzione al polimorfismo

Esempio di pessimo uso di RTTI

```
class Figura{};
class Cerchio:public Figura{};
class Triangolo: public Figura{};
class Rettangolo:public Figuar;
void Rotate(const Figuar &f) {
        if(typeid(f) == typeid(Cerchio))
                ;               //Non fare niente
        else if(typeid(f) == typeid(Triangolo))
                ...;            //Applica la rotazione a "f"come Triagolo
        else if(typeid(f) == typeid(Rettangolo))
                ...;            //Applica la rotazione a "f"come Rettangolo
}
```

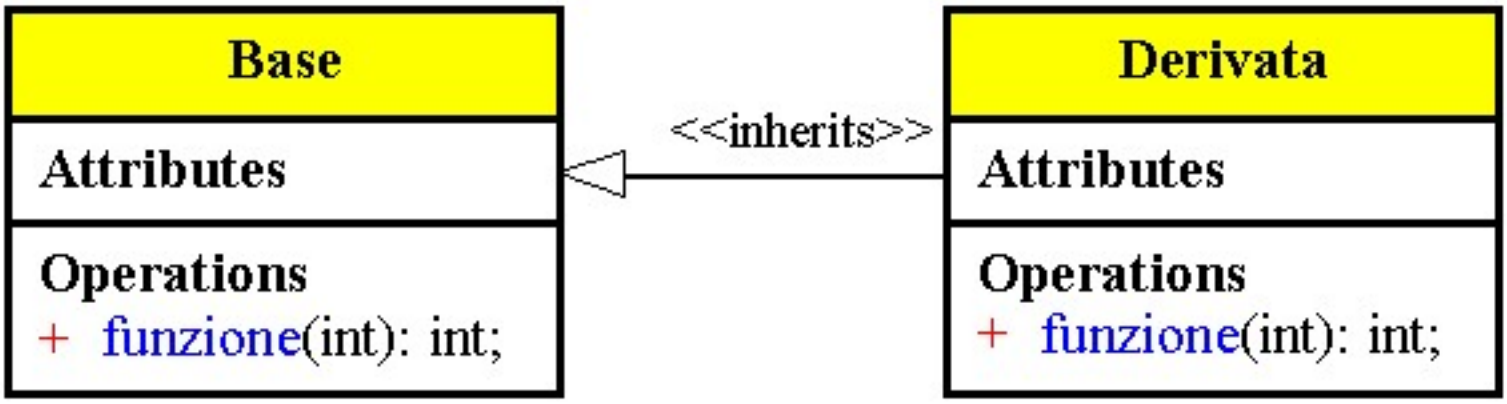
Ritorna

```
/* Nometipo.cpp
 * primo esempio dell'uso della classe Type_info
 */
#pragma hdrstop
#include <condefs.h>
#include <iostream.h>
// include le definizioni per RTTI
#include <typeinfo.h>
//---------------------------------------------------------------------------
#pragma argsused
class UnaClasse
{};
void Attesa(char *);
int main(int argc, char* argv[]) {
  int numero = 27;
  UnaClasse unOggetto;
  cout << "Variabile: numero, Valore: " << numero
       << ", Tipo: " << (typeid(numero)).name() << endl;
  cout << "Variabile: unOggetto, Tipo: "
       << (typeid(unOggetto)).name() << endl;
  Attesa("terminare");
  return 0;
}
void Attesa(char * str) {
  cout << "\n\n\tPremere return per " << str;
  cin.get();
}
```

Ritorna

```
/* Checkt.cpp
 * usa RTTI per controllare run-time a cosa punta un puntatore
 */
#pragma hdrstop
#include <condefs.h>
#include <iostream.h>
#include <typeinfo.h>
//---------------------------------------------------------------------------
#pragma argsused
class Base {
public:
  virtual int funzione (int n)  {
    // codice senza senso: abbiamo bisogno di una funzione virtuale
    return n*2;
  }
};

class Derivata : public Base {
  int funzione (int n) {
    return n*3;
  }
};
void Attesa(char *);
int main(void) {
  Base* pBase = new Base;
  Derivata* pDerivata = new Derivata;
  cout << "Il programma inizia..." << endl;
  if (typeid(*pBase) == typeid(Derivata))
    cout << "Il puntatore a base punta a un oggetto derivato" << endl;
  else
    cout << "Il puntatore a base punta veramente ad una base" << endl;
  delete pBase;
  pBase = pDerivata;
  cout << "Assegnamenti..." << endl;
  if (typeid(*pBase) == typeid(Derivata))
    cout << "Il puntatore a base punta a un oggetto derivato" << endl;
  else
    cout << "Il puntatore a base punta veramente ad una base" << endl;
  delete pDerivata;
  Attesa("terminare");
  return 0;
}
void Attesa(char * str) {
  cout << "\n\n\tPremere return per " << str;
  cin.get();
}
```

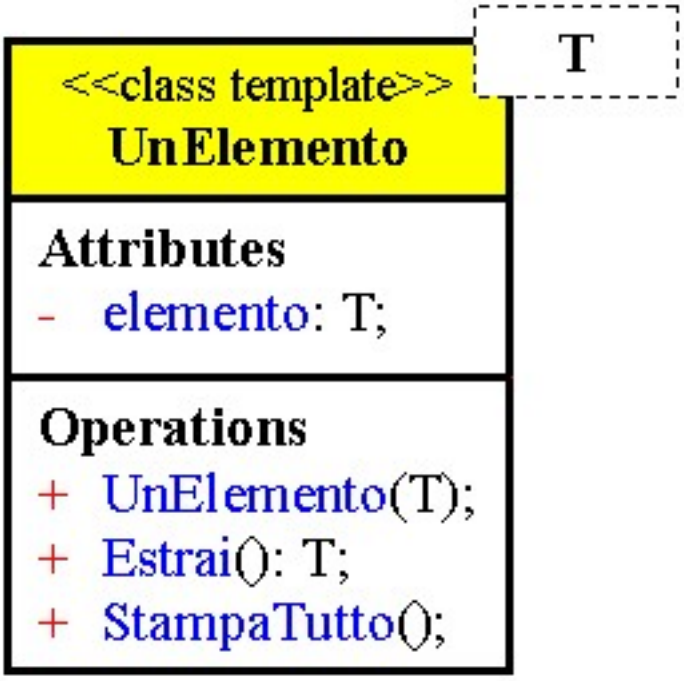
Diagramma U. M. L. delle classi



Ritorna

```
/* Temprtti.cpp
 * un breve esempio di RTTI con templates
 */
#pragma hdrstop
#include <condefs.h>
#include <iostream.h>
#include <typeinfo.h>
#include <cstring.h>
//---------------------------------------------------------------------------
#pragma argsused
template <class T>
class UnElemento {
public:
  UnElemento(T t): elemento (t) {}
  T Estrai() {
    return elemento;
  }
  void StampaTutto() {
    cout << "Elemento <" << elemento << "> di tipo "
         << typeid (elemento).name() << endl;
    // o: typeid(T).name()
  }
private:
  T elemento;
};
void Attesa(char *);
int main(int argc, char* argv[]) {
  UnElemento<int> uno(27);
  uno.StampaTutto();
  UnElemento<char> due('x');
  due.StampaTutto();
  UnElemento<int> tre('x');
  tre.StampaTutto();
  UnElemento<string> quattro("Tipo run-time di un template");
  quattro.StampaTutto();
  Attesa("terminare");
  return 0;
}
void Attesa(char * str) {
  cout << "\n\n\tPremere return per " << str;
  cin.get();
}
```

Diagramma U. M. L. delle classi



Spazio dei nomi

La parola chiave namespace offre un modo per evitare la collisione dei nomi tra librerie, esso pone i nomi in spazi distinti.
Esempio di definizione di un namespace:

```
namespace Mialib {
        class Y {
                static int j;
        public:
                void g();
        };
        class Z;
        void f();
}                       // Non ci vuole il punto e virgola
```

I namespace possono essere nidificati
Esempio:

```
namespace SistGenerale {
        class Cliente;
        class Prodotto;
        namespace SottoSistema {
                class Fattura;
        }
}
```

operatore scope resolution

Per accedere alle dichiarazioni all'interno di un namespace bisogna specificare l'accesso con l'operatore di scope resolution
Esempio:

```
Mialib::Y *y =new Mialib::Y;
y->g();
Mialib::Y::j = 8;
class Mialib::Z {
        int v;
public:
        void h();
};
```

Parola chiave using

Con la parola chiave using si può importare un namespace
Esempio:

```
using namespace Mialib;
Y *y = new Y;
Y::j = 13;
```

La parola chiave using non può essere usata all'interno di una classe
Esempio:

```
class ListaClienti {
        usinf namespace Fatture;       // Errore
        …
};
```

Ambiguità

L'operatore di scope resolution può essere usato per risolvere le ambiguità
Esempio:

Vendite.h

```
namespace Commerciale {
        class Cliente;
        class Prodotto;
}
```

Fatturaz.h

```
namespace Fatture {
        class Cliente;
        class Fattura;
}
```

Esempio.cpp

```
void prova() {
        using namespace Commerciale;
        using namespace Fatture;
        Prodotto prod;
        Commerciale::Cliente cl1;
        Fatture::Cliente cl2;
}
```

Anche la parola chiave using può essere usata per risolvere le ambiguità
Esempio:

Esempio.cpp

```
void prova() {
        using namespace Commerciale;
        using namespace Fatture;
        using Commerciale::Cliente;
        Prodotto prod;
        Cliente cl1;
        Fatture::Cliente cl2;
}
```

Estensione di un namespace

La ridefinizione di un namespace ne estende il contenuto
Esempio:

header1.h

```
namespace Mialib {
        extern int gb;
        int funz();
}
```

header2.h

```
namespace  Mialib {
        float read();
}
```

Non è possibile estendere un namespace dopo aver usato la direttiva using
Esempio:

```
namespace Mialib {
        extern int gb;
        int funz();
}
using Mialib::funz();
namespace Mialib {
        float read();
}
```

Sinonimi

Possono essere assegnati dei sinonimi ai namespace
Esempio:

```
namespace SistemaGeneraleDiFatturazione {
        class Cliente;
        class Fattura;
}
namespace SF = SistemaGeneraleDiFatturazione;
```

namespace anonimi

Un namespace può essere anonimo, risultano utili per limitare la visibilità di variabili al solo file dove sono definiti.
Esempio:

prova.cpp

```
        namespace {
                double pi;
        }
        int funz() {
                double pi = 3.14.159;
                cout << pi << endl;
        }
```

Nuova sintassi per i cast

La sintassi per i cast è stata estesa per evidenziare le violazioni ai tipi di dati.

dynamic_cast

consente di eseguire il downcasting in modo sicuro.
dynamic_cast e puntatori dynamic_cast<T*>(p);
L'operatore dynamic_cast converte il puntatore p al tipo T* solo se p è un T o una classe derivata da T
Se il puntatore non è convertibile dynamic_cast restituisce 0 (null)
Esempio:

```
void funz(Base *base) {
        Derivata *derivata;
        if((derivata = dynamic_cast<Derivata*>(base)) != 0)
                ……
}
```

dynamic_cast e riferimenti dynamic_cast<T&>(ref);
L'operatore dynamic_cast converte il riferimento ref al tip T& solo se ref è un T o una classe darivata da T
Se il riferimento non è convertibile dynamic_cast lancia l'eccezione bad_cast
Esempio:

```
void funz(Base &base) {
        try {
                Derivata &derivata = dynamic_cast<Derivata &>(base);
                        …….
        } catch(bad:cast &e)
                        …….
  }
```

Il dynamic_cast è appropriato nel caso di ereditarietà multipla
Esempio:

```
class BB {
public:
        virtual void f() {}
};
class B1 : virtual public BB {};
class B2 : virtual public BB{};
class MI : public B1, public B2 {};

BB* bbp = new MI;                       // Upcast
MI* mip = dynamic_cast<MI*>(bbp);       // Lavora correttamente
```

static_cast

static_cast è usato quando il cast è affidabile.
Esempio:

```
int i;
long l;
double d;
l = static_cast<long>(i);
d = static_cast<double>(i);
```

Può causare perdita di informazione, ma evita warning del compilatore.
Esempio:

```
i = static_cast<int>(l);
i = static_cast<int>(d);
```

static_cast deve essere usato per convertire void* ad altri tipi
Esempio:

```
void *p = &i;
double *dp = static_cast<double*>(p);
```

static_cast non può essere usato per fare conversioni non legittime.
Esempio:

```
char *pc = static_cast<char*>(&l);  // Errore di compilazione
```

static_cast potrebbe essere usato al posto di dynamic_cast quando è fondamentale l'efficienza ma meno sicuro.

const_cast

const_cast è usato per rimuovere l'attributo const o volatile
Esempio:

```
const char *s = "Luca";
char *p = const_cast<char *>s);
```

const_cast è usato quando una funzione deve trattare parametri const in modo non const
Esempio:

```
void funz(const Cliente &cli) {
        Cliente &c = const_cast<Cliente&>(cli);
        c.MetodoNonCost();
}
```

La parola chiave mutable è una valida alternativa quando il const_cast è applicato a data members, mutable indica che il data member può essere modificato anche se l'oggetto è cosnt.
Esempio:

```
class string {
        char *buf;
        mutable int ncopie;
private:
        string(const string &rhs) {
                rhs.ncopie++;
        }
};
```

A volte const_cast deve essere complementato da altri cast
Esempio:

```
voif funz(const Base *pb) {
        Base *pbase = const_cast<Base *>(pb);
        Derivata *pder = dynamic_cast<Derivata *>(pbase);
}
```

reinterpret_cast

reinterpret_cast deve essere usato in tutte le situazioni non supportate dagli altri cast, permette qualunque tipo di cast ma è il meno sicuro di tutti i tipi di cast.
Esempio:

```
class Base {};
class Derivata: public Base {};
class Altra {};
const Base *pbase = new Derivata;
Derivata *pder = reinterpret_cast<Derivata *>(pbase);      // Rimuove const e esegue
downcast
Altra *pa = reinterpret_cast<Altra *>(pbase);                    // Non ha senso ma è
ammesso
```

reinterpret_cast non dovrebbe mai essere usato!

Ritorna

```
/* Rtshape.cpp
 * Conta le figure
 */
#pragma hdrstop
#include <condefs.h>
#include <iostream.h>
#include <time.h>
#include <typeinfo.h>
#include "arrdin.h"
#include "figure.h"
//---------------------------------------------------------------------------
#pragma argsused
void Attesa(char *);
int main(int argc, char* argv[]) {
  Arrd<figura> fig;
  time_t t;
  srand((unsigned)time(&t));
  const mod = 6;
  for(int i=0; i<rand()%mod; i++)
    fig.Add(new rettangolo);
  for(int j=0; j<rand()%mod; j++)
    fig.Add(new ellisse);
  for(int k=0; k<rand()%mod; k++)
    fig.Add(new cerchio);
  int Ncerchi  = 0;
  int Nellissi = 0;
  int Nrettangoli = 0;
  int Nfigure  = 0;
  for(int u=0; u < fig.Cont(); u++) {
    fig[u]->draw();
    if(dynamic_cast<cerchio*>(fig[u]))
      Ncerchi++;
    if(dynamic_cast<ellisse*>(fig[u]))
      Nellissi++;
    if(dynamic_cast<rettangolo*>(fig[u]))
      Nrettangoli++;
    if(dynamic_cast<figura*>(fig[u]))
      Nfigure++;
  }
  cout << endl << endl
       << "Cerchi     = " << Ncerchi     << endl
       << "Ellissi    = " << Nellissi    << endl
       << "Rettangoli = " << Nrettangoli << endl
       << "Figure     = " << Nfigure     << endl
       << endl
       << "cerchio::quanti()    = " << cerchio::quanti()    << endl
       << "ellisse::quanti()    = " << ellisse::quanti()    << endl
       << "rettangolo::quanti() = " << rettangolo::quanti() << endl
       << "figura::quanti()     = " << figura::quanti()     << endl;
  Attesa("terminare");
  return 0;
}
void Attesa(char * str) {
  cout << "\n\n\tPremere return per " << str;
  cin.get();
}
```

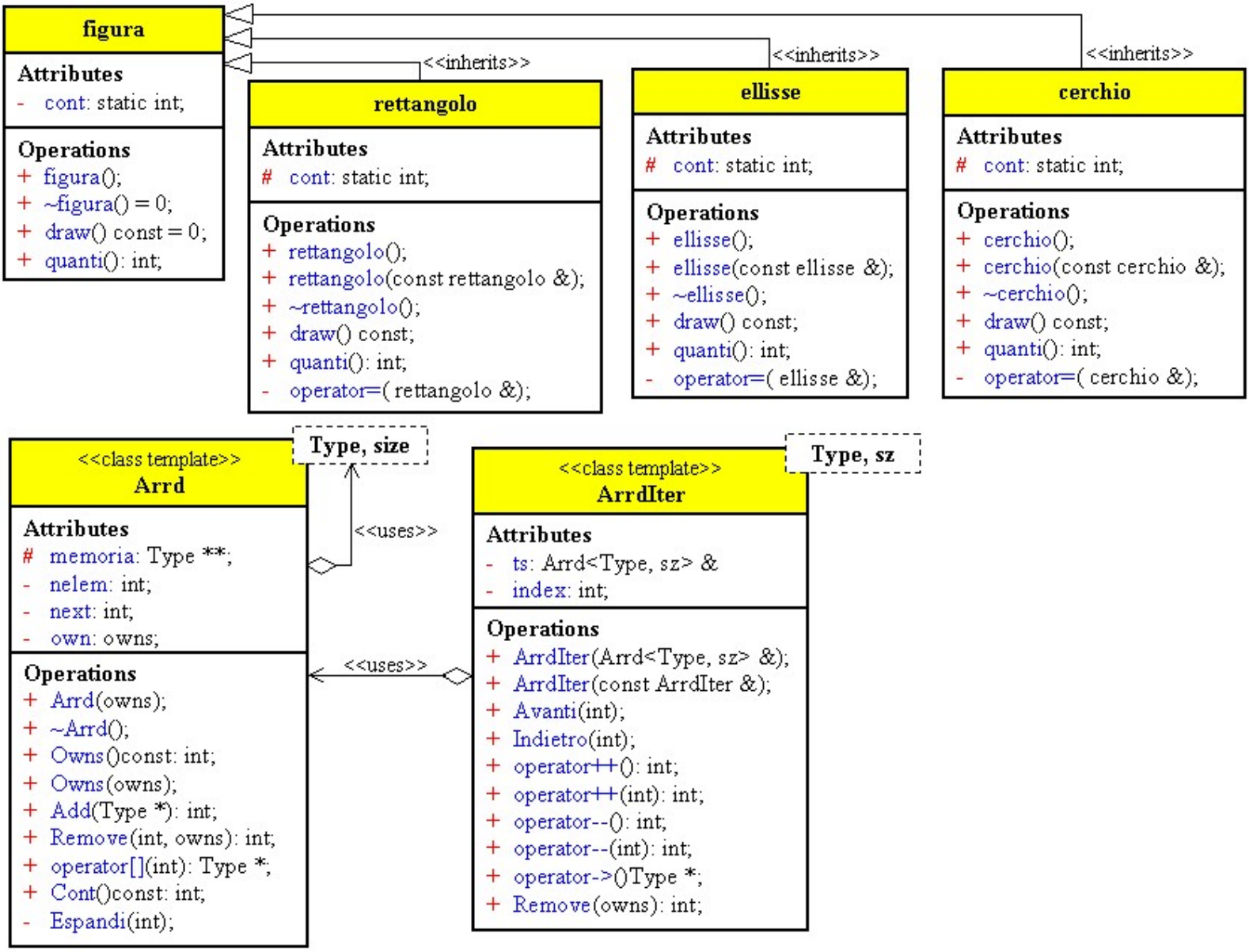
```
/* figure.h
 * figure geometriche
 */
#ifndef FIGURE_H
#define FIGURE_H
class figura {
protected:
public:
  figura() { cont++; }
  virtual ~figura() = 0 { cont--; }
  virtual void draw() const = 0;
  static int quanti() { return cont; }
private:
  static int cont;
};

int figura::cont = 0;

class rettangolo : public figura {
protected:
  static int cont;
public:
  rettangolo() { cont++; }
  rettangolo(const rettangolo &) { cont++; }
  ~rettangolo() { cont--; }
  void draw() const {
    cout << "rettangolo::draw()" << endl;
  }
  static int quanti() { return cont; }
private:
  void operator=(rettangolo&);  // Disabilita l'assegnamento
};
int rettangolo::cont = 0;

class ellisse : public figura {
protected:
  static int cont;
public:
  ellisse() { cont++; }
  ellisse(const ellisse &) { cont++; }
  ~ellisse() { cont--; }
  void draw() const {
    cout << "ellisse::draw()" << endl;
  }
  static int quanti() { return cont; }
private:
  void operator=(ellisse&);  // Disabilita l'assegnamento
};
int ellisse::cont = 0;

class cerchio : public figura {
protected:
  static int cont;
public:
  cerchio() { cont++; }
  cerchio(const cerchio &) { cont++; }
  ~cerchio() { cont--; }
  void draw() const {
    cout << "cerchio::draw()" << endl;
  }
  static int quanti() { return cont; }
private:
  void operator=(cerchio&);  // Disabilita l'assegnamento
};
int cerchio::cont = 0;
#endif  // FIGURE_H
```

```c
/* Chekerr.h
 */
#ifndef CHEKERR_H
#define CHEKERR_H
#include <stdio.h>
#include <stdlib.h>
#include <assert.h>
inline void chek_error(int val, const char *msg) {
  if(!val) {
    fprintf(stderr, "Errore: %s\n", msg);
#ifdef NDEBUG
    exit(1);
#endif
  }
}

#define chekerr(expr, msg) \
{  chek_error((expr) ? 1 : 0, msg);  \
        assert(expr);  }

#define chekerrmem(expr) \
        chekerr(expr, "Fuori memoria")

#define chekerrfile(expr) \
        chekerr(expr, "Non si puo` aprire il file")

#endif //CHEKERR_H
```

```
/* Arrdin6.h
 * Array dinamico creato a tempo di esecuzione
 * Utilizzo dei template
 */
#ifndef ARRDIN7_H
#define ARRDIN7_H
#include <stdlib.h>
#include "Chekerr.h"
enum owns {no = 0, si = 1, Default};
// Dichiarazione
template<class Type, int sz> class ArrdIter;
template<class Type, int size = 20>
class Arrd {
protected:
  Type** memoria;          // Memoria dinamica
public:
  Arrd(owns Owns = si);        // Costruttore
  ~Arrd();               // Distruttore
  int Owns() const {return own; }
  void Owns(owns newOwns) { own = newOwns; }
  int Add(Type *element);
  int Remove(int index, owns d = Default);
  Type * operator[](int index);
  int  Cont() const {return next;}          // Conta gli elementi
  friend class ArrdIter<Type, size>;
private:
  int nelem;               // Numero di elementi
  int next;                // Prossimo spazio disponibile
  owns own;
  void Espandi(int increm = size);      // Espande il vettore
};

template<class Type, int sz = 20>
class ArrdIter {
public:
  ArrdIter(Arrd<Type, sz> &TS) : ts(TS), index(0) {}
  ArrdIter(const ArrdIter &rv) : ts(rv.ts), index(rv.index) {}
  // Spostamento dell'iteratore
  void Avanti(int spost) {
    index += spost;
    if(index >= ts.next)
      index = ts.next - 1;
  }
  void Indietro(int spost) {
    index -= spost;
    if(index < 0)
      index = 0;
  }
  int operator++() {
    if(++index >= ts.next)
      return 0;
    return 1;
  }
  int operator++(int) { return operator++(); }
  int operator--() {
    if(--index < 0)
      return 0;
    return 1;
  }
  int operator--(int) { return operator--(); }
  operator int() {
    return index >= 0 && index < ts.next;
  }
  Type * operator->() {
    Type *t = ts.memoria[index];
    if(t) return t;
    chekerr(0, "ArrdIter::operator-> return 0");
    return 0;
  }
  // Rimuove l'elemento corrente
  int Remove(owns d = Default) {
    return ts.Remove(index, d);
  }
private:
  Arrd<Type, sz> &ts;
  int index;
};
```

```
template<class Type, int sz>
Arrd<Type, sz>::Arrd(owns Owns) : own(Owns) {
  nelem = 0;
  memoria = 0;
  next = 0;
}
// Distruzione degli oggetti contenuti
template<class Type, int sz>
Arrd<Type, sz>::~Arrd() {
  if(!memoria) return;
  if(own == si)
    for(int i=0; i < Cont(); i++)
      delete memoria[i];
  free(memoria);
}

template<class Type, int sz>
int Arrd<Type, sz>::Add(Type *element) {
  if(next >= nelem)     // Vi e' sufficiente spazio?
    Espandi();
  // Copia l'elemento nella memoria
  memoria[next++] = element;
  return (next - 1);
}

template<class Type, int sz>
int Arrd<Type, sz>::Remove(int index, owns d) {
  if(index >= next || index < 0)
    return 0;
  switch(d) {
    case Default:  if(own != si) break;
    case si:       delete memoria[index];
    case no:       memoria[index] = 0;      // La posizione e' vuota
  }
  return 1;
}

template<class Type, int sz> inline
Type * Arrd<Type, sz>::operator[](int index) {
  assert((index >= 0) && (index < next));
  return memoria[index];
}

template<class Type, int sz>
void Arrd<Type, sz>::Espandi(int increm) {
  void *v = realloc(memoria, (nelem + increm)*sizeof(Type *));
  chekerrmem(v);
  memoria = (Type **)v;
  nelem += increm;
}
#endif   // ARRDIN7_H
```

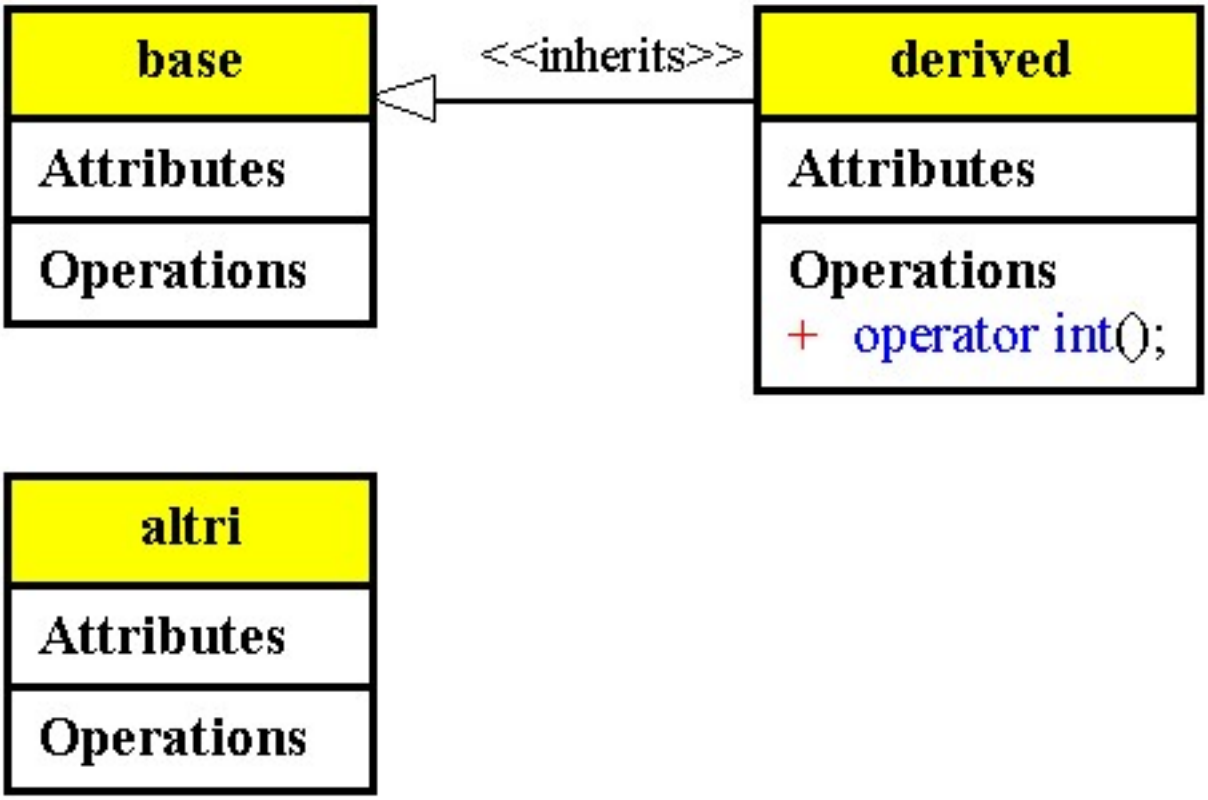
Diagramma U. M. L. delle classi



Ritorna

```
/* Statcast.cpp
 * static_cast   esempi
 */

#pragma hdrstop
#include <condefs.h>
#include <iostream.h>

//---------------------------------------------------------------------------
#pragma argsused
class base { };
class derived : public base {
public:
  operator int() { return 1; }
};
void funz(int ii) {cout << "ii =  " << ii << endl;}
class altri{};
void Attesa(char *);
int main(void) {
  int i = 45;
  long l;
  float f;
  // conversione implicita
  l = i;
  f = i;
  l = static_cast<long>(i);
  f = static_cast<float>(i);
  f *= 3.63;
  cout << "f =   " << f << endl;
  // conversione con perdita di informazione
  i = l;
  i = f;
  // elimina i warnings
  i = static_cast<int>(l);
  i = static_cast<int>(f);
  char c = static_cast<char>(i);
  cout << "i =  " << i << "     c = " << c << endl;
  // conversioni con void *
  void *vp = &i;
  // Attenzione operazione pericolosa
  float *fp = (float *)vp;
  fp = static_cast<float*>(vp);
  cout << "i =  " << i << "    *fp =  "  << *fp << endl;
  // conversioni implicite
  derived d;
  base *bp = &d;     // Upcast: normale e OK
  bp = static_cast<base*>(&d);       // piu` esplicita
  int x = d;         // conversione automatica
  x = static_cast<int>(d);           // piu` esplicita
  funz(d);           // conversione automatica
  funz(static_cast<int>(d));         // piu` esplicita
  // Navigazione nella gerarchia
  // Piu` efficiente del dynamic_cast ma meno sicuro
  derived *dp = static_cast<derived*>(bp);
  // Indica un errore
  // altri *ap = static_cast<altri*>(bp);
  // Non indica l'errore
  altri *ap2 = (altri *)bp;
  Attesa("terminare");
  return 0;
}
void Attesa(char * str) {
  cout << "\n\n\tPremere return per " << str;
  cin.get();
}
```

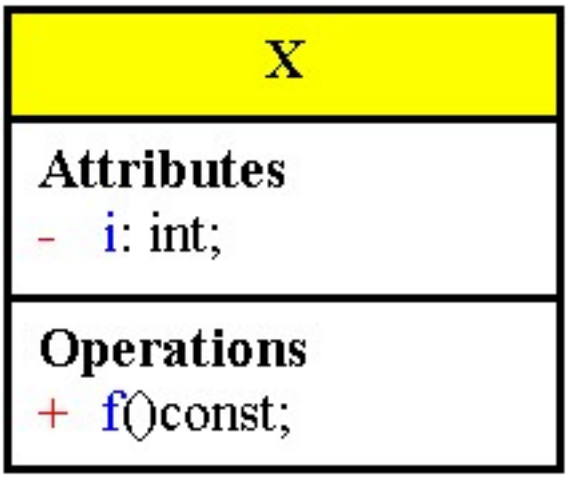
Diagramma U. M. L. delle classi



Ritorna

```
/* Constcst.cpp
 * const_cast   esempi
 */

#pragma hdrstop
#include <condefs.h>
#include <iostream.h>

//---------------------------------------------------------------------------
#pragma argsused
class X {
public:
  void f() const {
    (const_cast<X*>(this))->i = 1;
    // j = 3;
  }
private:
  int i;
  // mutable int j; // Approccio migliore (questa versione non e` aggiornata)
};
void Attesa(char *);
int main(void) {
  const int i = 3;
  const int *c = &i;
  // *c = 7;                  // Errore
  int *j;
  // j = (int *)&i;                          // Forma deprecabile
  j = const_cast<int *>(&i); // Da preferire
  *j = i + 5;                // Non modifica i
  *j = i * (*j);
  cout << "i = " << i << "      *j = " << *j << endl;
  // Non e`possibile utilizzare piu` cast simultaneamente
  // long* l = const_cast<long*>(&i);           // Errore
  volatile int k = 0;
  int *u = const_cast<int *>(&k);
  *u = 9;
  cout << "k = " << k << "      *u = " << *u << endl;
  Attesa("terminare");
  return 0;
}
void Attesa(char * str) {
  cout << "\n\n\tPremere return per " << str;
  cin.get();
}
```

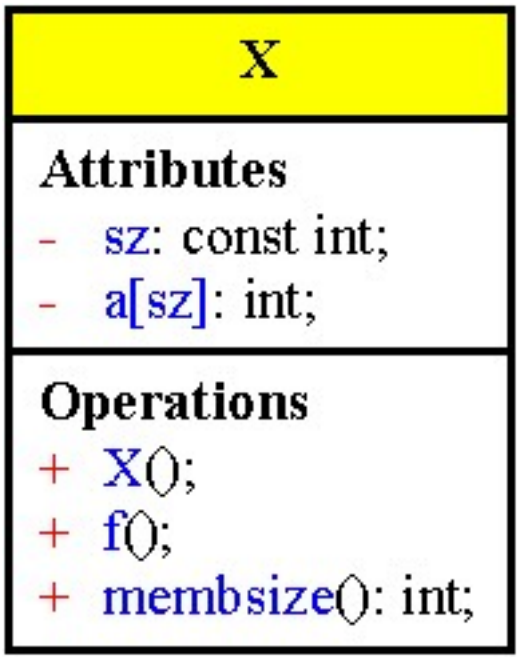
Diagramma U. M. L. delle classi



Ritorna

```
/* Reinterp.cpp
 * reinterpret_cast esempio
 */

#pragma hdrstop
#include <condefs.h>
#include <iostream.h>
#include <fstream.h>
//---------------------------------------------------------------------------
#pragma argsused
ofstream out("reinterp.out");
class X {
public:
  X() { memset(a, 0, sz*sizeof(int)); }
  virtual void f() {}
  int membsize() { return sizeof(a); }
  friend ostream& operator<<(ostream &os, const X& x) {
    for(int i=0; i<sz; i++)
      os << x.a[i] << ' ';
    return os;
  }
private:
  enum {sz = 5};
  int a[sz];
};
void Attesa(char *);
int main(void) {
  X x;
  out << x << endl;
  int *xp = reinterpret_cast<int*>(&x);
  xp[1] = 47;
  out << x << endl;
  X x2;
  const vptr_size = sizeof(X) - x2.membsize();
  long l = reinterpret_cast<long>(&x2);
  l += vptr_size;
  xp = reinterpret_cast<int*>(l);
  xp[1] = 47;
  out << x2 << endl;
  Attesa("terminare");
  return 0;
}
void Attesa(char * str) {
  cout << "\n\n\tPremere return per " << str;
  cin.get();
}
/* Non e` sicuro assumere che i dati dell'oggetto inizino all'indirizzo
 * iniziale dell'oggetto, difatti questo compilatore pone la VPTR all'inizio
 * dell'oggetto
 */
```

Diagramma U. M. L. delle classi



STL Standard Template Library

E' una libreria di modelli; fornisce una serie di parti che possono essere unite per eseguire le operazioni richieste da un programma essa comprende:

contenitori	classi che contengono collezioni di oggetti
iteratori	che servono a percorrere il contenuto delle classi contenitore
algoritmi	che definiscono procedure generiche
oggetti funzione	che incapsulano funzioni
allocatori	che incapsulano diverse strategie di allocazione della memoria

Contenitori

Contenitori sequenziali contengono una collezione di oggetti secondo un preciso ordine lineare

vector	sono gli array dinamici
list	sono liste doppiamente linkate
deque	hanno le prestazioni di un array nell'accesso indicizzato ma permettono l'inserzione e l'estrazione rapida sia in testa che in coda

Contenitori associativi permettono la ricerca veloce sugli elementi contenuti secondo una chiave

set	implementa sequenze ordinate ad accesso associativo rapido senza elementi ripetuti
multiset	come i set ma ammettono elementi ripetuti
map	contengono coppie chave-valore non ripeture
multimap	come le map ma con coppie ripetute

Contenitori adattatori riduce o modifica l'interfaccia di un'altra classe contenitore

stack	strettamente LIFO
queue	strettamente FIFO
priority_queue	mantiene gli elementi in un ordine stabilito

Iteratori

Un iteratore rappresenta una posizione in una struttura dati, corrisponde al concetto di cursore su una struttura dati e generalizza l'uso di un puntatore.
STL classifica gli iteratori secondo una gerarchia di requisiti, ogni tipo di iteratore deve supportare una lista di operazioni.

input	può leggere un elemento alla volta solo nella direzione in avanti
output	può scrivere un elemento alla volta solo nella direzione in avanti
forward	combina le caratteristiche degli iteratori di input e output
bidirectional	come forward più la possibilità di muoversi all'indietro
random	come bidirectional più la possibilità di saltare ad una distanza arbitraria

Gli iteratori di output sono utilizzati per scrivere valori su contenitori illimitati (es. output standard)
E’ impossibile trovare la distanza fra due iteratori di output

Il tipo di iteratore è associato ai vari contenitori nel modo seguente:

Contenitore	Iteratore
vector	random
deque	random
list	bidirectional
multiset	bidirectional
set	bidirectional
multimap	bidirectional
map	bidirectional
stack	nessuno
queue	nessuno
priority_queue	nessuno

Algoritmi

Gli algoritmi sono detti generici perché sono fatti in modo da poter funzionare sia sulle strutture dati definite da STL sia sulle strutture dati primitive.
Gli algoritmi non agiscono direttamente sulle strutture dati ma tramite entità chiamate iterarori.
L'idea base di STL è di implementare gli algoritmi sotto forma di template e in modo che possano funzionare su tutti i tipi che soddisfano alcuni requisiti minimali.
Tutti gli algoritmi sono indipendenti dall'implementazione di una particolare struttura dati e sono caratterizzati solo dal tipo di iteratori su cui operano.
Possiamo suddividerli nelle seguenti categorie:

Categoria	**Esempi**
non su contenitori	swap, min, max, advance
di scansione non mutativa	for_each, equal, count
di scansione mutativa	transform, copy, fill, replace, remove
di ricerca	find, min, max, search
spostamenti, permutazioni e partizioni	swap, reverse, rotate
di sort	sort, parzial_sort
sui set	set_union, set_intersection, set_difference
operazioni su heap	
numerici	accumulate, prodotto intero, somma parziale

Oggetti funzione

Un oggetto funzione è un tipo di dato che definisce l'operatore (), esso è l'astrazione di un puntatore a funzione
Tipi di oggetti funzione:

- Generatori: non prendono alcun argomento
- Funzioni unarie: accettano un argomento, possono anche restituire un valore
- Funzioni binarie: accettano due argomenti, possono anche restituire un valore

Quando un oggetto funzione restituisce un valore di tipo bool si chiama predicato.

Allocatori

Gli allocatori sono oggetti che incapsulano la gestione della memoria dinamica di un contenitore.
Esempio:

template <class T, template<class U>class Allocator=allocator>class vector {….};

Allocator è un template di classe destinato ad esere parametrizzato secondo il tipo degli oggetti contenuti nella struttura dati.
Per comodità Allocator ha un default fornito da STL di nome allocator, che gestisce la memoria dinamica usando gli operatori new e delete
Ogni contenitore contiene un oggetto allocatore che alloca la memoria richiesta dal contenitore.
Inoltre gli allocatori incapsulano le informazioni pertinenti al modello di memoria dell'ambiente operativo: dimensioni dei puntatori, tipo risultante dalla differenza fra due puntatori.

Note libreria STL

I contenitori sequenziali sono più adatti per accedere agli elementi in modo sequenziale utilizzando l'operatore [ ] o gli iteratori
I contenitori associativi sono ottimizzati per fornire un accesso diretto ai propri elementi basandosi sul valore della chiave.
Le classi contenitore richiedono che gli elementi siano in grado di:

- Deve avere un costruttore standard
- Deve avere un costruttore di copia
- Deve avere un operatore di assegnamento
- Quando si richiede di eseguire confronti si deve definire l'oveloading degli operatori di confronto