

INFORMATICA

definizione:

è lo scienze che studia come viene rappresentata l'informazione e degli algoritmi che le trasformano servendosi di un calcolatore

algoritmico:

seguente, precise e finite, di operazioni per la realizzazione di un computo, queste operazioni possono essere di tipo:

- 1) sequenziale: realizzano una impone ovvero
- 2) condizionale: controllano una condizione
- 3) iterativa: ripetono le operazioni finché non è verificata una condizione

le proprietà di un algoritmo sono:

- 1) eseguibile: ogni azione eseguita in un tempo finito
- 2) non ambiguo: azione interpretabile dell'operatore
- 3) finito: il numero di azioni deve essere finito
- 4) corretto: poter trattare l'insieme di dati in ingresso
- 5) efficiente: se lo fa nel modo più efficiente

INFORMAZIONE

Si presenta in forma astratta, seguente 8 bit (Byte) e bisogna rappresentarla.
Bit (Binary digit) è l'unità di misura elementare dell'informazione, ossia i valori 0, 1.

quante informazioni può essere contenuta?:

in n bit l'informazione corrisponde alle possibili combinazioni di 0, 1
in n caselle, 2^n se $M=3$, ci saranno 2^3 caselle (bit) (le possibili combinazioni) da n spazi

fare approfondimenti ASCII

CODIFICA ASCII

La codifica ascii (american standard code for information interchange) è una tabella di codifica da 8 bitte + 1 bit per rappresentare caratteri alfamumerici, simboli e caratteri di controllo

- supporto (standard): 128 caratteri ($2^7=128$), 7 bit
- supporto (extra): 256 caratteri ($2^8=256$), 8 bit
- supporto (unicode): 65536 caratteri ($2^{16}=65536$), 16 bit → non supportate da c++

Tabelle

000	001	010	011	100	101	110	111	<i>punteggio circa</i>
NUL	DLE	SP	0	@	P	'	p	0000
SOH	XON	!	1	A	Q	a	q	0001
STX	DC2	"	2	B	R	b	r	0010
ETX	XOFF	#	3	C	S	c	s	0011
EOT	DC4	\$	4	D	T	d	t	0100
ENQ	NAK	%	5	E	U	e	u	0101
ACK	SYN	&	6	F	V	f	v	0110
BEL	ETB	'	7	G	W	g	w	0111
BS	CAN	(8	H	X	h	x	1000
HT	EM)	9	I	Y	i	y	1001
LF	SUB	*	:	J	Z	j	z	1010
VF	ESC	+	:	K	[k	{	1011
FF	FS	,	<	L	\	l		1100
CR	GS	-	=	M]	m	}	1101
SO	RS	.	>	N	^	n	-	1110
SI	US	/	?	O	_	o	DEL	1111 <i>ultimo</i>

RAPPRESENTAZIONE DEI NUMERI NATURALI

I numeri possono essere rappresentati con diverse **basi**:



base 10:

- numeri 0-9
- rappresentazione posizionale

$$(123)_{10} = 1 \cdot 10^2 + 2 \cdot 10^1 + 3 \cdot 10^0$$



base 2:

- numeri 0-1
- rappresentazione posizionale

$$(11001)_2 = 1 \cdot 2^4 + 1 \cdot 2^3 + 0 \cdot 2^2 + 0 \cdot 2^1 + 1 \cdot 2^0$$

OSS:

- 1) ogni cifra ha associato un **peso** intero
- 2) amplificato poi dalla **base**

$$N = \sum_{i=0}^{p-1} a_i B^i = a_{p-1} B^{p-1} + a_{p-2} B^{p-2} + \dots + a_1 B^1 + a_0 B^0$$

N = numero da rappresentare

B = base

a_i = cifra del numero

i = posizione delle cifre

p = lunghezza sequenza di cifre

se **B = 2** (codice binario)

Potenza di due	Valore in base dieci	Denominazione
2 ⁰	1	
2 ¹	2	
2 ²	4	
2 ³	8	
2 ⁴	16	
2 ⁵	32	
2 ⁶	64	
2 ⁷	128	
2 ⁸	256	
2 ⁹	512	
2 ¹⁰	1024	1 Kilo
...	...	
2 ²⁰	1048576	1 Mega
2 ³⁰	1073741824	1 Giga
2 ³²	4294967296	4 Giga

per rappresentare un numero in base 2, è necessario porre 1 seguito da **n** zero

$$2^5 = 1 \underbrace{00000}_{n}$$

Da base B a base 10:

dato uno qualsiasi seguente di cifre, è necessario applicare il procedimento mentrato per trasformare un numero in base 10, dividendo ogni cifra il suo pero precedente oss

Da base 10 a base B:

Esempio: da base 10 a base 2

$$N = 23$$

inizio

QUOZIENTE	RESTO	Rappresentazione
23	-	
11	1	$11 \cdot 2 + 1$
5	1	$((5 \cdot 2) + 1) \cdot 2 + 1$
2	1	$((((2 \cdot 2) + 1) \cdot 2) + 1) \cdot 2 + 1$
1	0	$((((1 \cdot 2) + 0) \cdot 2) + 1) \cdot 2 + 1$
0	1	$((((0 \cdot 2) + 1) \cdot 2 + 0) \cdot 2 + 1) \cdot 2 + 1$

ri amplifica

$(10111)_2$
 $a_4 a_3 a_2 a_1 a_0$

che in base dieci vale $1 \cdot 2^4 + \dots + 1 = (23)_{dieci}$, cvd

In generale:

Inizio $q_0 = N$

QUOZIENTE	RESTO
$q_0 = N$	-
$q_1 = q_0 / \beta$	a_0
$q_2 = q_1 / \beta$	a_1
$q_3 = q_2 / \beta$	a_2
$q_4 = q_3 / \beta$	a_3
$q_5 = q_4 / \beta$	a_4
$q_6 = q_5 / \beta$	a_5
$q_7 = 0$	a_6

fine

$$N = a_6 \cdot \beta^6 + a_5 \cdot \beta^5 + a_4 \cdot \beta^4 + a_3 \cdot \beta^3 + a_2 \cdot \beta^2 + a_1 \cdot \beta^1 + a_0 \cdot \beta^0$$

esercizio:

$$(87)_{10} \quad 8\text{-bit}$$

uso div mod:

$$\begin{array}{r} 87 \\ \hline 1 | 43 | 2 \\ 1 | 21 | 2 \\ 1 | 10 | 2 \\ 1 | 5 | 2 \\ 1 | 2 | 2 \\ 1 | 1 | 2 \\ 1 | 0 | 2 \\ \hline & & 0 \end{array}$$

1) mi occorre quattro volte
2) contro i costi del prezzo
3) ne avranno un bit in meno o
 $(01010111)_2$

$$(0101101)_2 \quad (?)$$

$$\begin{array}{r} 0101 \\ 1101 \\ \hline \end{array} \equiv 5$$

$$(-107)_{10} \quad 8\text{-bit}$$

$$2^8 - 10001 = 149$$

$$\begin{array}{r} 149 \\ \hline 1 | 74 | 2 \\ 1 | 34 | 2 \\ 1 | 18 | 2 \\ 1 | 9 | 2 \\ 1 | 4 | 2 \\ 1 | 2 | 2 \\ 1 | 1 | 2 \\ 1 | 0 | 2 \\ \hline & & 0 \end{array}$$

10010101

01101001

ES SOMMA

$$A = (01000111)_2$$

$$B = (10011101)_2$$

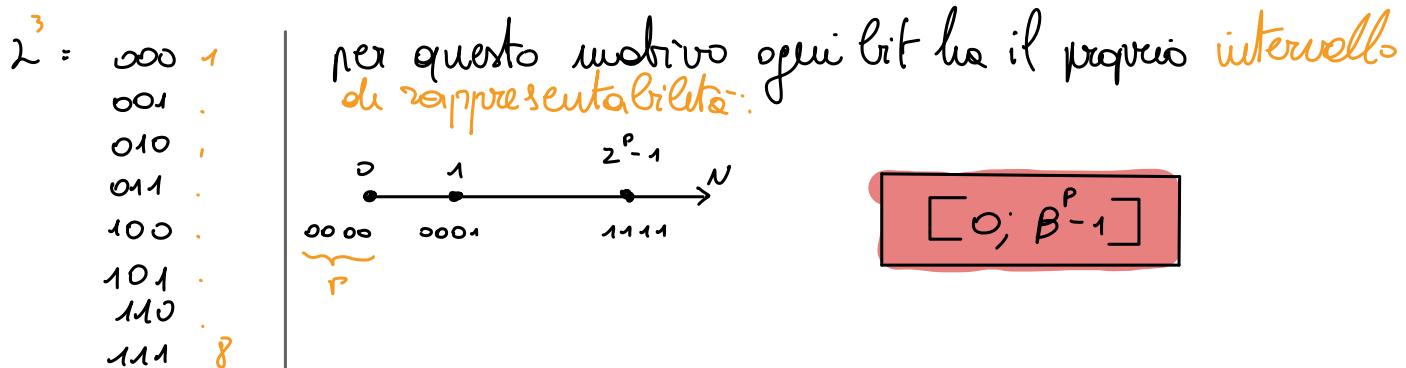
$$\begin{array}{r} 01000111 + \\ 10011101 = \\ \hline 11100100 \end{array}$$

$$\begin{array}{r} 01100111 + \\ 10011100 = \\ \hline 100000011 \end{array}$$

overflow!

intervallo di rappresentabilità:

come abbiamo detto, il p bit corrisponde al numero di codelle che al numero di cifre all'interno di questo, visto che deve rappresentare le possibili combinazioni:



overflow: dato un codicudo con un numero finito di bit, si perde quando il risultato non rientra nel range di rappresentabilità.

ES: $\underline{101100} + \underline{111010} =$

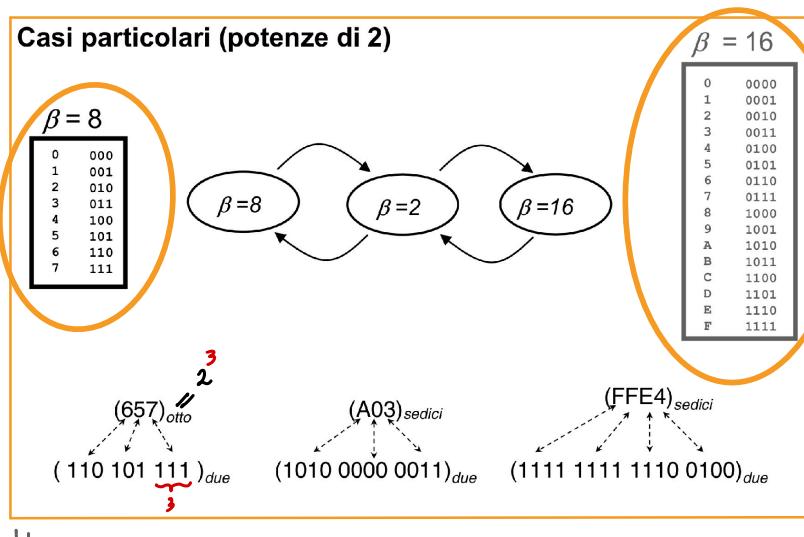
il risultato richiede 7 bit

$$\begin{array}{r} 101011 \\ + 111010 \\ \hline 100101 \end{array}$$

scambio di base:

- in generale, è necessario passare per base 10 e poi fare divisione
- per trovarlo in base 2^m partendo da una base che è un multiplo

Casi particolari (potenze di 2)



ES: $(\underline{\underline{010011}})_2 \rightarrow (\underline{\underline{23}})_8$

- per convertire un codice di base 2 in base 2^m è necessario formare dei gruppi di m cifre e dare corrispondenza. NB: per completarci il gruppo è possibile aggiungere all'inizio uno 0.
- per convertirlo in base 2, riportare il gruppo di m cifre (esponente base di parentesi) corrispettivo della cifra ridotta.

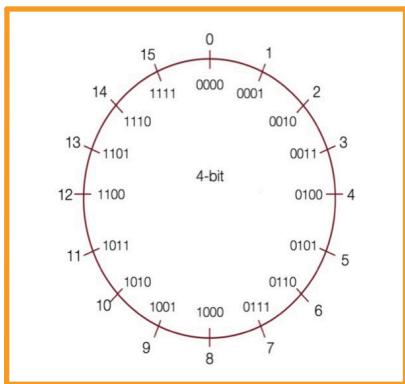
ri rappresentazione numeri \mathbb{N} in c++:

si possono definire m.

- 1) 16 bit
- 2) 32 bit
- 3) 64 bit

unsigned short int
unsigned int
unsigned long int (potrebbe battersi su 32)

■ se manda in stampa un valore che non rientra nell'intervallo di rappresentazione, mi verrà restituita la differente che esce da tale intervallo partendo dall'ultimo



RAPPRESENTAZIONE N. INTERO

Per rappresentare i numeri interi possiamo utilizzare 3 metodi:

modulo e segno:

Questo metodo utilizza:

- 1 bit per il segno

- 0 se positivo
- 1 se negativo

- $p-1$ bit per rappresentare il numero in valore assoluto $abs(a)$

ES: $P=4$: $\begin{array}{l} s = +3 \Rightarrow 0011 \\ s = -3 \Rightarrow 1011 \end{array}$

per il procedimento inverso invece:

$$[-(2^{P-1}-1), + (2^{P-1}-1)]$$

- si guardano le prime cifre
- si moltiplica il valore assoluto dell'intero

!= Verifica rappresentabilità

Rappresentazione complemento a due:

importante: complemento a due permette di eseguire somma e sottrazione utilizzando le regole del bit, dunque non necessita di un'altra circolazione

Io il 99% delle CPU utilizza questo metodo

In questo procedimento viene:

- 1) controllare le prime cifre per vedere il segno (vedi rapp. precedente)
- 2) applicare la condizione:

per contrarii

$$\rightarrow - (a >= 0) ? (abs(a)) : -(2^P - abs(a))$$

ma 0 rapp.

$$[-(2^{P-1}), + (2^{P-1}-1)]$$

se $a > 0$ allora restituisce il valore assoluto, altrimenti calcola quanto manca a completere 2^P

NB: se è positivo, ti sta moltiplicando la rappresentazione del numero, se è negativo, ti sta moltiplicando quanto manca $ES: -1 = 1111$

Rappresentazione con Bias

Questo metodo permette di rappresentare i numeri negativi come positivi effettuando una **trasformazione** equivalente a un **intervallo**:

per numero $\rightarrow A = d(t) \left(2^{P-1} - 1\right)$ per 0

$$\boxed{\left[(2^{P-1} + 1), + (2^{P-1}) \right]}$$

VIRGOGLA FISSA

Utilizzo per rappresentazione numeri decimali.

principio di funzionamento

- Inizia un numero finito di bit per le parti intera e una per la parte decimale
- la virgola non si rappresenta
- le parti intere si rappresentano con le tecniche note, le parti frazionarie invece così...

rappresentazione parte decimale

Sia $I_{(1)}$ la parte intera e $F_{(1)}$ la parte frazionaria, si segue questo processo iterativo:

$$5,6875 \quad F_0 \neq 0$$

$$F_0 = 0,6875$$

$$F_1 = F_0 \cdot 2 = 1,375$$

$$F_2 = F_1 \cdot 2 = 0,75$$

$$F_3 = F_2 \cdot 2 = 1,5$$

$$F_4 = F_3 \cdot 2 = 1$$

ho superato 1? sì, 1

ho superato 1? no, 0

ho superato 1? sì, 1

ho superato 1? sì, 1

la parte frazionaria sarà 1011

rappresentazione numero

se riduiamo di rappresentare 1 su 16 bit e f su 5 bit:

$$\pm, 0000000101,10110$$

- centrosovrapposizione = $1 \cdot 2^{-1} + 0 \cdot 2^{-2} + 1 \cdot 2^{-3} + 1 \cdot 2^{-4}$

$$(0.6875)_{10} \Leftrightarrow (0.1011)_2$$

errore di truncamento

quando si incalza un periodico decimale, si perde
all'ultimo bit rappresentabile

- il numero da rappresentare r diventa r' ,
- l'errore di truncamento diventa $z(z\text{ iniziale}) - z'$ (calcolato in β_m)

LINGUAGGIO DI PROGRAMMAZIONE

per definire sintassi e semantica di un altro linguaggio, utilizzando un metalinguaggio, in notazione BNF

- metal. sintassi = insieme di notazioni (non ambigue), spiegate con parole semplici del linguaggio naturale
- metal. semantica = si ricorre al linguaggio naturale, perché utilizzando un metal. matematico risulterebbe molto complesso

notazione

- 1) basata su grammatiche BNF (metalinguaggio per notarsi linguaggi formali)
- 2) terminologia propria
- 3) simboli con regole semplificate (prefissi basic)
- 4) diverse organizzazione delle categorie sintattiche (espressioni, dichiarazioni)

regole

Le regole descrivono le varie categorie sintattiche,

- ① • categorie sintattiche = le categorie che le regole descrive, in corsivo
- costrutti metalinguaggio = elementi usati per indicare come le varie parti delle regole devono essere combinate, saltuariamente
- simboli terminali = componenti che appaiono nel programma (int, float), caratteri normali.

ES: dichiarazione → tipo identificatore ;

① cosa descrivono

② tipo = elemento sintassi (int, float)

identificatore = nome variabile

③ notare come può essere utilizzata

NB: quando ci sono più modi per descrivere una regola, le forme alternative possono essere disposte su righe separate, oppure utilizzando uno o più indicatori per indicare le possibili scelte

elementi di una categoria sintattica

- optionali = contrassegnati con suffisso opt
- ripetuti più volte = con introduzione con categorie sintattiche oppure
 - richiede qualunque di elementi

some-element-seq

- lista, virgole

some-element-list

SINTASSI C++

- costituite da sequenze di parole (tokens)
- parole delimitate da spazi bianchi (white space)

parole

costituire dei seguenti caratteri:

- token-character
- digit : 1, 2, 3 ...
- letter : a, b, c, D ...
- special : !, . / , ^

spazi bianchi

- carattere spazio
- carattere tabulazione
- nuova riga

commenti

- sequenze di parole a blocco racchiuso fra /* e */
- caratteri // a fine riga

elementi lessicoli:

- 1) identificatori = congegni nome univoco (int x)
- 2) parole chiave = parole con un significato preciso nel linguaggio (while, for..)
- 3) espressione letterali = denotano valori costanti ("reali", "caratteri", "stringhe")
- 4) separatori = segni di interruzione o raggruppamento.... (; , ())
- 5) operatori = caratteri che denotano operazioni
proprietà:

1) posizione rispetto agli operandi :
• prefissi + 5
• postfissi x ++
• infixi 4 + 5

2) ordine : numero di operandi

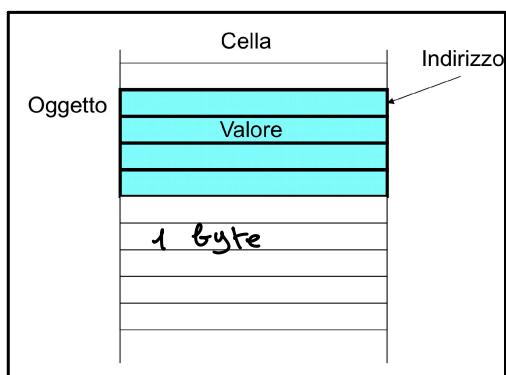
3) precedente : priorità degli operatori : $3 + 5 * 2$

4) associatività : ordine con cui vengono eseguiti

stessi operatori
• da sx $(1 + 2 + 3)$
• da dx $(1 = 1 = 1)$

OGGETTI

gruppi di celle consecutive che vengono considerate come unica cella informatica



Attributi

- indirizzo
- valore

contenuti e variabili

- il contenuto può variare o non rispettivamente
- l'indirizzo non cambia

settore di memoria

appelti al momento della dichiarazione:

- 1) quanto occupa in memoria
- 2) quali valori può assumere
- 3) quali sono le operazioni permesse

settore di operazione

comprende:

- 1) simbolo
- 2) entità (funzione, funzione)
- 3) posizione del simbolo rispetto agli operandi
- 4) associazioni
- 5) precedenze

Tipi di un oggetto

FONDAMENTALI



ENUMERAZIONE

Tipi predefiniti:

- tipo intero (int) e tipo naturale (unsigned);
- tipo reale (double); 64 bit, (float) 32 bit
- tipo booleano (bool);
- tipo carattere (char).

DERIVATI

- ↓
- a partire da tipi predefiniti;
 - strutture dati più complesse



- riferimenti
- puntatori
- array
- strutture
- classi
- unioni

Tipo INZERO

i tipi interi sono, come in matematica, i numeri positivi e negativi non decimali

Proprietà -

1) valgono le operazioni elementari:

- cambio segno, resto
- $+, -, \cdot, \div,$

2) per le diverse gestione del range valgono:

- lung int = suffisso "L"
- short int
- int ottale (base 8) = prefisso "0"
- int esadecimale (base 16) = prefisso "0X"

Accorgimenti

l'insieme dei numeri reali lo possiamo intendere come un sottousieme dei numeri razionali, ed il range di rappresentabilità dipende dal numero di bit disponibili.

Per questo possiamo parlare di:

- overflow: quando il numero che si vuole rappresentare esce fuori dal range di rappresentabilità
- underflow: quando il numero che si vuole rappresentare e' troppo vicino allo 0

TIPO UNSIGNED

uso

unsigned sta per nulla segno, e' inoltre possibile che vogliano introdurre, quindi non dovranno rappresentare i negativi aumentando il campo di rappresentabilità.

caso int

- 1) quando si usa con int = suffisso "U"
- 2) quando si usa con short int = mille
- 3) quando si usa con long = VL

- N.B.:
- int puo' essere omesso, perché - solt' intero
 - long int puo' non avere il suffisso

osservazioni

- 1) campo di rappresentazione 2^{n-1}
- 2) tipo di operazioni semplici
- 3) contenuto delle celle visto come serie di bit

TIPO REALE

Si compone di parte intera e parte frazionaria:

intera frazionaria

10. 56 → rappresentata in un'inglese linea (double)

operazioni

Le operazioni sui reali e sugli interi si eseguono con gli stessi operatori.

TIPO BOOL

restituiscono costanti predefinite False, True (rispettivamente 0, 1)

operazioni

	OR logico o disgiunzione			
&&	AND logico o congiunzione			
!	NOT logico o negazione			
p	q	p q	p && q	!p

false	false	false	false	true
false	true	true	false	true
true	false	true	false	false
true	true	true	true	false

NB: nel OR, AND e stampa definito tre parentesi

cout << (p1(q)) << endl;

implicazioni logiche

$p \rightarrow q$ "Se p allora segue q"

l'implicazione è F solo se, a seguito di un vero c'è un falso

p	q	$p \rightarrow q$!p	$!p \parallel q$

false	false	false	true	true
false	true	true	true	true
true	false	false	false	false
true	true	true	false	true

operatori di confronto

- operatori di uguaglianza (precedute più bassa)
- operatori di relazione

==	uguale
!=	diverso
>	maggiore
>=	maggiore o uguale
<	minore
<=	minore o uguale

operatori bit a bit

- | OR bit a bit
- & AND bit a bit
- ^ OR esclusivo bit a bit
- ~ complemento bit a bit
- << traslazione a sinistra
- >> traslazione a destra

almeno 1
entrambi
solo 1
opposto
sposta i bit di posizione, lasciando 0

a	b		&	^	~a	~b
0	0	0	0	0	1	1
0	1	1	0	1	1	0
1	0	1	0	1	0	1
1	1	1	1	0	0	0

maschere

utilizzando una "maschera", una stringa di numeri binari, è possibile alterare e lasciare invariati i valori a nostra piacimento

- 0 reset:

- AND: è necessario creare una variabile formata da zeri in corrispondenza dei bit da restituire, e 1 di quelli che vogliamo lasciare invariati

ES: 01010101
00000010
00000111

- 1 reset.

- OR: è necessario creare una variabile formata da 1 in corrispondenza dei bit da fornire, e 0 di quelli che vogliamo lasciare invariati

ES: 01010101
00000010
01010111

- XOR: è necessario creare una variabile formata da 1 in corrispondenza dei bit da invertire, e 0 di quelli che vogliamo lasciare invariati

ES: 01010101
00001100
01010011

regole del cortocircuito

Le regole del cortocircuito si applicano alle operazioni logiche $\&\&$, $\| \|$. permette di ridurre il numero di espressioni valutate all'interno di un'operazione logica

- $\&\&$: in un'espressione del tipo $A \&\& B$, se A è falso B non viene mamente valutata (notare che A,B non sono bool ma int, per avere fatti debbono essere zero), se A è vero viene valutata anche l'altra
- $\| \|$: in un'operazione del tipo $A \| B$, se A è vera B non viene mamente valutata, se è falso valuta anche l'altra perché gliene basta 1 vera

ES: double num, den, soglia;
cin >> num >> den;

if (den != 0 $\&\&$ num/den > soglia) Si verifica denominatore = 0

TIPO CARATTERE

Il tipo carattere è rappresentato dal tipo char, utilizzato per memorizzare i singoli caratteri.

Rappresentazione e dimensione

Un carattere fra apici rappresenta un numero intero (1 byte) secondo la codifica ASCII (in ordine alfabetico)

il valore può essere espresso in:

- ottale = \....
- esadecimale = \x
- decimale

caratteri di controllo

combinazioni speciali che iniziano con una barra invertita

- nuova riga (LF)	\n
- tabulazione orizzontale	\t
- ritorno carrello (CR)	\r
- barra invertita	\
- apice	\'
- virgolette	\"

TIPO ENUMERATION

Il tipo enum è un insieme definito di costanti (che rappresentano numeri interi) associate dal programmatore ad un identificatore detto enumeratore.

- intervallo finito di costanti
- rappresentano informazioni non numeriche

operazioni:

- 1) confronti: $(LUN == MER)$
- 2) operazioni su interi: definite come generiche operazioni

esempio

```
#include <iostream>
using namespace std;
int main()
{
    enum Giorni {LUN,MAR,MER,GIO,VEN,DOM};
    Giorni oggi = MAR;
    oggi = MER;
    int i = oggi; // ② conversione implicita
    // oggi = MER-MAR; // ERRORE! MER-MAR->intero
    // oggi = 3; e' di tipo enum // ERRORE! 3 costante intera
    // oggi = i; di tipo int, contiene 2
    cout << int(oggi) << endl; // 2
    cout << oggi << endl; // 2, conv. implicita

    enum {ROSSO, GIALLO, VERDE} semaforo;
    semaforo = GIALLO;
    cout << semaforo << endl; // 1

    enum {INIZ1=10, INIZ2, INIZ3=9, INIZ4};
    cout << INIZ1 << 't' << INIZ2 << 't';
    cout << INIZ3 << 't' << INIZ4 << endl;
}
```

N.B.: posso far uscire la sequenza non per forza da 0

```
2
2
1
10   11   9   10
```

CONVERSIONI ESPULSE

dunque non esiste una conversione implicita, per effettuare una conversione da un tipo ad un altro, si usa

- static - cast<in cosa> cosa
- in cosa = (in cosa) cosa

- solo se entro le conversioni implicite inverse

esempio

```
#include <iostream>
using namespace std;
int main()
{
    enum Giorni {LUN,MAR,MER,GIO,VEN,SAB,DOM};
    int i; Giorni g1 = MAR, g2, g3;
    i = g1; restituisce l'indirizzo intero
    g1 = static_cast<Giorni>(i); venga convertito il valore di i (1) in giorno (MAR) (che è 1)
    g2 = (Giorni)i; stesso modo // cast
    g3 = Giorni(i); notazione funzionale

    cout << g1 << '\t' << g2 << '\t' << g3 << endl;
    int j = (int) 1.1; voglio il valore in int // cast, 1
    float f = float(2); // notazione funzionale
    cout << j << '\t' << f << endl;
}
```

1	1	1
1	2	

CONVERSIONI IMPULZE

concetto

quando due variabili di tipo diversi vengono combinate in espressione, il sistema esegue conversioni implicite sostituendo un tipo componibile

esempio

int J;

float f = 2,5F \rightarrow specifica float e non double, meno memoria

J = f + 1 // 5 numero decimale troncato

osservazioni

perdite informazione

- 1) nella conversione da double a int \Rightarrow perdite di informazione
- 2) interi grandi quando convertiti in float \Rightarrow perdite informazione poiché float perdono l'informazione utilizzando il formato floating-point che per dover rappresentare anche i decimali, arrotonda le cifre meno significative

conversioni

- 1) operandi dello stesso tipo ma lunghezze diverse, il più piccolo verrà esteso
- 2) operandi di variabili diverse, intero + reale, restituire valore reale

assegnamento

- 1) a una variabile reale può essere assegnato un valore intero
- 2) a una variabile di tipo intero può essere assegnato un valore reale corretto, esattamente, binario,

es: int x = 5
float y = x // 5.0

OGGETTI COSTANTI

Si usa per far sì che il valore di quelle variabili rimanga costante

- quando definiscono una variabile, mettiamo const prima
- se sempre inizializzata

here pagine concernanti

size of

valore in byte ha grandezza di una variabile

- `char` : 1 byte
- `short` : 2 byte
- `int` : 4 byte
- `long` : 4 byte
- `unsigned char` : 1 byte
- `unsigned short` : 2 byte
- `unsigned int` : 4 byte
- `unsigned long` : 4 byte
- `float` : 4 byte
- `double` : 8 byte
- `long double` : 12 byte

ES. `char c = 2`

- `cout << sizeof (c) << endl;`
- `cout << sizeof (cchar) << endl` NO

STRUTTURA DI UN PROGRAMMA

basic-main-program

int main () compound-statement inizio struttura programma
compound-statement blocco di codice racchiuso fra graffe
{ statement-seq } sequenze di istruzioni

statement unità base del codice eseguibile:

declaration-statement - dichiarazioni funzioni e variabili

definition-statement - definizione (atribuzione valore)

expression-statement - istruzione espressiva (operazioni...)

structured-statement - istruzioni strutturate, controllano il flusso (if, while, for...)

jump-statement - inseriscono un salto nel flusso di controllo

labeled-statement - contengono etichette

Istruzioni di dichiarazione/definizione:

declaration-statement

definition-statement

} dichiarazione e definizione seguono le norme standard

hanno la forma vista in precedenza.

Simboli introdotti dal programmatore: simboli (funzioni e variabili)

- devono essere dichiarati/definiti prima di essere usati;
- non è necessario che le dichiarazioni/definizioni precedano le altre istruzioni.

strutture espressione

Sintassi:

basic-expression-statement

expr opt ; istruzione espressiva (optional), cioè "potrebbe esserci anche solo"; "per rispettare sintassi"

expr

se combinato [term] expr infix-binary-op expr istruzione espressiva con operatore binario infisso

term parte dell'espressione

primary-exp

pre-fixed-unary-op expr operazione unaria prefissa (operatore precede) "++e"

expr post-fixed-unary-op operazione unaria postfissa (operatore segue) "e ++"

primary-exp

literal valore costante, un carattere o stringa

identifier identificatore: nome di variabile o funzione

(expr) sotto-espressione tra parentesi

Esempio:

- formata da letterali, identificatori, operatori, ecc., e serve a calcolare un valore;
- opzionale, perché in certi casi può essere utile usare una istruzione vuota (che non compie alcuna operazione) indicando il solo carattere ";".

Esempi di istruzioni espressione:

```
5;            // letterale seguito da ;
a;            // nome di variabile seguito da ;
-a;           // operatore unario prefisso
a+b;          // operatore binario infisso
(a+7)*-b;    // combinazione dei precedenti
```

INCREMENTO / DECREMENTO

di seguito alcune regole per il pre e post incremento:

int $x = 5$ // 6
int $y = x++$ // 5

int $x = 5$ // 6
int $y = ++x$ // 6

espressioni di eseguimento

l'eseguimento è trattato come un'expressione, quindi può compiere all'interno di una espressione

funzionamento

- 1) valutazione a destra dell'eseguimento: rappresenta l'expressione che viene calcolata (costante, variabile)
- 2) eseguibile alla variabile a sinistra: deve essere una locazione di memoria modificabile in cui viene memorizzato il valore, rappresenta il risultato dell'eseguimento (quando si legge da dx e va)

sintassi

- basic: variable-name = expression
- basic-recurrence: variable-name = variable-name \oplus expression
variable modificata operatore espressione valutata

$$\underline{\text{ES}} \quad x = x + 5$$

- basic-compound: variable-name \oplus = expression | equivalente, miglior leggibilità -

$$\underline{\text{ES}} \quad x + = 5$$

associatività e precedenza

- 1) Fattori = incremento / decremento postfisso (vari, da dx)
 - " prefisso
 - not logico
 - meno numeri
 - più numeri

- 2) termini:
 - moltiplicativi
 - additivi (binari, da dx)
 - di relazione
 - di ugualanza
 - logici (ff, 11) (da dx)
 - eseguimento (da dx)

NB . parentesi toglie modificano le precedenze

l'operazione termina appena è possibile stabilire la veridicità, "regole del cortocircuito"

operazioni

Non solo permette operazioni aritmetiche, di confronto, di assegnamento

- nei cicli di concatenazione è possibile effettuare confronti, vengono confrontati gli "avvertiti"

es

{

int u [5]

int v [6]

if (u == v)

OPERATORE CONDIZIONALE

struttura e spiegazione

$e_1 ? e_2 : e_3$

\downarrow \downarrow
if...then... otherwise

se e_1 è vero fa e_2 , altrimenti fa e_3

utilizzo

e' comune quando devi voler fare una condizione come vero o falso ed hai varie scelte solo due opzioni

OPERATORE VIRGOLA

operatore binario infino, associativo da sinistra con priorità inferiore a tutti

struttura e spiegazione

exp1, exp2

vengono valutate (calcolate) le espressioni, l'operatore restituisce come risultato l'ultima espressione

utilizzo

per poter inserire più espressioni dove la grammatica ne prevederebbe una sola

esempio

1) int a = 2

int b = 3

d = (b e_1 , e_2)

cout << a << endl; // 5

cout << b << endl; // 4 (l'espressione viene calcolata)

2) int a = 2

int b = 3

(a = b++) 5 (essa priorita', viene prima assegnamento)

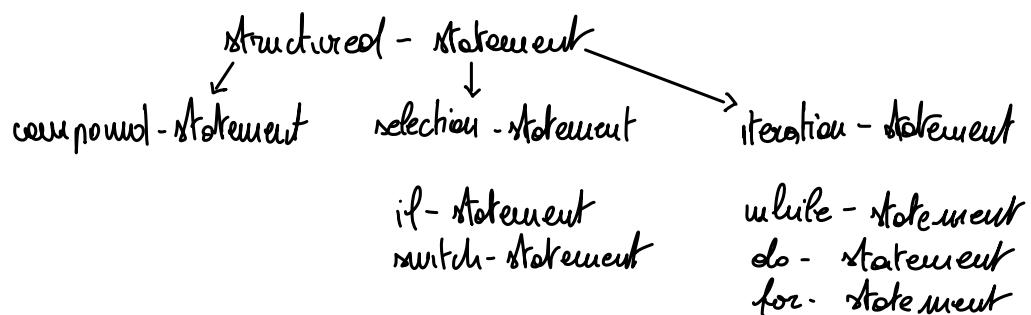
cout << a << endl; // 3 (primo assegnamento poi incremento)

cout << b << endl; // 4

ISTRUZIONI STRUTTURATE

con queste istruzioni possiamo controllare il flusso di programma, grazie a condizioni, decisioni e iterazioni.

Schemi



ISTRUZIONE COMPOSTA

con i delimitatori { } è possibile trasformare più istruzioni in un'unica istruzione, ovunque il codice prevede un'istruzione

ISTRUZIONI CONDIZIONALI

IF

Struttura

- 1) if (selection) statement
- 2) if (selection) statement else statement

Utilizzo

quando il compilatore legge il codice, nel caso 1 se trova le condizioni verificate esegue un'azione, altrimenti va avanti. Nel caso 2 se le condizioni non sono vere fa qualcosa altrimenti prima di eseguire l'azione mette un'altra

caso particolare:

- quando abbiamo 3 optioni, poniamo che:

if 1)

else if 2)

else 3)

e' importante mettere che prima un valore potrebbe soddisfare le prime due condizioni, else sta dicendo "io ci metto solo se le prime non mi basta"

- if "semidato":

- 1) if 1) il 2 if viene valutato solo se il primo e' soddisfatto.
- 2) if 1)

ES: int eta = 18;
bool permesso = true;

- 1) if (eta >= 18) // maggiore ?
 - 2) if (permesso) // ha permesso ?
cout << "accesso consentito" << endl;
else cout << "accesso negato" << endl;

notare che:

- if (bool):

un if (cond1 || cond2) si verifica solo se il valore di bool e' true

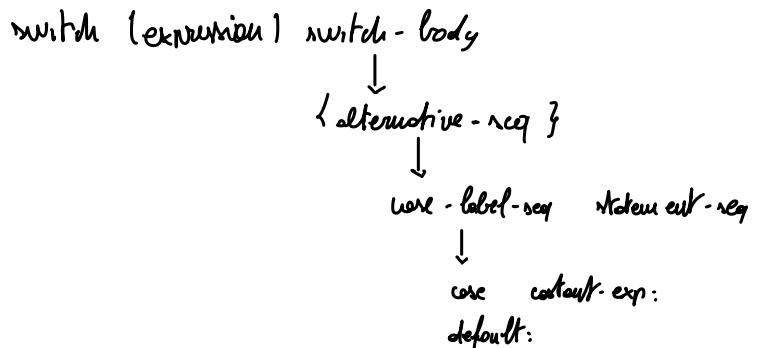
SWITCH

struttura

```

switch-statement
  switch ( expression ) switch-body
  switch-body
    { alternative-seq }
  alternative
    case-label-seq statement-seq
  case-label
    case constant-expression :
  default :

```



esecuzione

- 1) viene valutata l'espressione
- 2) viene eseguita l'alternativa con l'etichetta in cui compare il valore calcolato (un blocco di codice può avere identificativi che più etichette) **ES**:
 case 1:
 case 2:
 constant-expression
- 3) viene valutata (o eseguita) l'ultima alternativa nel caso in cui non venga individuata l'etichetta **default**

se il valore assegnato non corrisponde a nessuna dei case, default indica l'opzione alternativa, se default c'è ovvero non viene eseguita nessuna istruzione

osservazioni

- 1) l'espressione è costituita da una variabile e valori diversi (int, char, enum...)
- 2) l'espressione case individua le alternative deve contenere una costante (int, char, A, B, c) dello stesso tipo dell'espressione iniziale (poiché chi scrive l'alternativa si riferisce al tipo dell'espressione che è il case)

break

e' importante inserire il comando break dopo ogni alternativa, se non lo si fa il programma valuterà anche i case successivi e quello individuato

esempio

```

#include <iostream>
using namespace std;
int main()
{
    cout << "Seleziona un'alternativa" << endl;
    cout << "A - Prima Alternativa" << endl;
    cout << "B - Seconda Alternativa" << endl;
    cout << "C - Terza Alternativa" << endl;
    cout << "qualsiasi altro tasto per uscire" << endl;
    char c;
    cin >> c;
    switch (c)
    {
        case 'a': case 'A':
            cout << "Prima alternativa" << endl;
            break;
        case 'b': case 'B':
            cout << "Seconda alternativa" << endl;
            break;
        case 'c': case 'C':
            cout << "Terza alternativa" << endl;
    }
    // Manca il caso di default
    // Se non è una delle alternative, non scrive niente
}
return 0;

```

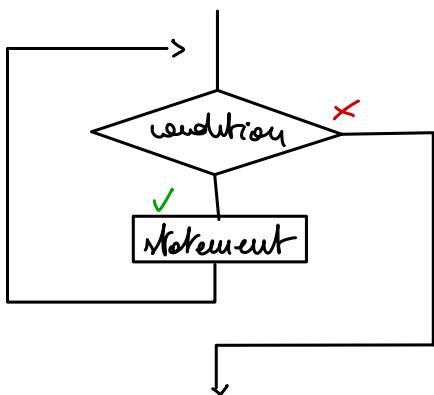
quando c'è di tipo char ricordati che ci sono anche le maiuscole

ITERATION

WHILE

Structure

while (condition) statement



finisce la condizione è verificata, inverte il ciclo

DO

Structure

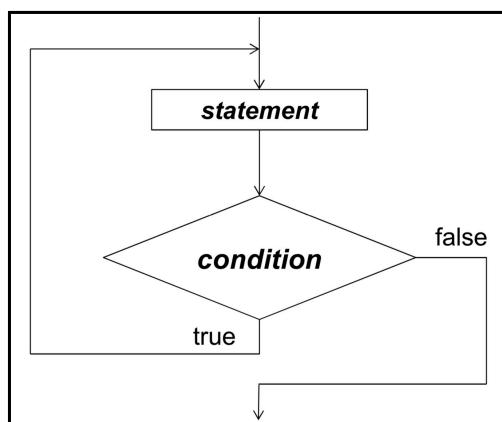
do statement , while (condition);

Execution

- 1) seguita l'istruzione fra do e while (corpo del do)
- 2) viene valutata la condizione

- se vero, l'istruzione viene ripetuta
- se falso, l'istruzione termina

NB: l'istruzione viene valutata sempre una volta



FOR

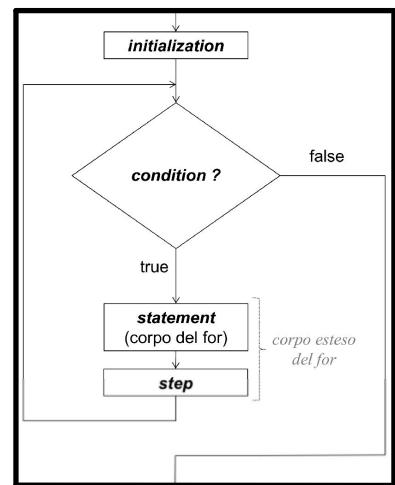
Structure

For (initialization; condition ; step statement)

1) initialization =
• espressione
• dichiarazione

2) condition =
• espressione booleana (v.o.F)

3) step statement =
• blocco di codice di ogni iterazione, ed
eventuale modifica
step



ISTRUZIONI SALTO

Jump - Statement

- 1) break - statement
- 2) continue - statement
- 3) goto - statement
- 4) return - statement

BREACK

soltanto all'istruzione immediatamente successiva al corpo del ciclo o dell'istruzione switch, lo si usa:

- nei cicli: per interrompere l'iterazione del ciclo
- negli switch: per prevenire il "fall-through", caso che verifichi eseguiti gli switch successivi a quel valore

CONTINUE

struttura

continue;

utilizzo

- serve la terminazione di un'iterazione del ciclo che le contiene
- soltanto parte del ciclo che voluti muovere la condizione \Rightarrow non esce dal ciclo

```
es   int i=0
      do {
          i++;
          if (i==2) {
              continue;
          }
          cout << i << endl;
      } while (i<5) // continuo a volutare da i=3
```

differenze continue e break

- 1) break: soltanto all'iterazione successiva al corpo del ciclo
- 2) continue: soltanto l'iterazione corrente ignorando le istruzioni

→ NB: può essere usato nell'ultima istruzione switch come re forse break

T. SACOPINI BOHM

terzi

questo terzetto afferma che qualsiasi algoritmo può essere rappresentato usando solo tre strutture di controllo fondamentali:

- sequenziale = istruzioni eseguite una dopo l'altra
- relazionale = varie condizioni e condizionali (if, switch)
- iterazionale = ripetere blocchi di codice (while, for, do-while)

quarti

prima di questo terzetto molti linguaggi facevano largo uso dell'istruzione goto che permetteva di saltare da una parte all'altra del programma, tuttavia l'uso eccessivo rendeva il codice difficile da leggere creando così quello che comunemente viene definito spaghetti-code

FUNZIONI

concetto

nella realizzazione di un programma spesso occorre che una stessa sequenza di operazioni debba essere ripetuta più volte eventualmente con argomenti diversi, dunque è necessario scrivere tale sequenza una sola volta sotto forma di funzione ed utilizzarla ove richiesto con un meccanismo di chiamata.

costrutti

1) **definizione**: viene indicata

intestazione
result-type identifier (argument-list)

corpo

- il nome della funzione
- quali sono gli argomenti con cui deve operare
- tipo del risultato
- quali sono le azioni da eseguire

N.B.: normalmente il compilatore richiede che una funzione sia stata preventivamente dichiarata, la definizione funge anche da dichiarazione perché contiene tutti gli elementi di quest'ultima

2) **chiamata**: viene indicata

specifica del nome

- i dati effettivi su cui la funzione opererà

proprietà di variabili e argomenti formali

1) **variabili**: • le variabili definite in una funzione sono locali alle funzioni stesse

2) **nomi**: • ogni funzione ha le proprie variabili locali e argomenti associati a oggetti distinti, se uno stesso nome viene usato in più puntelli si riferiscono a oggetti diversi
• il nome è "visibile" solo nelle rispettive funzioni

3) **chiamata**:

• quando una funzione viene chiamata, le variabili e gli argomenti formali vengono allocati in memoria

N.B.: org. formali-attuali:
• ~~numero~~ numero
• ~~ordine~~ ordine
• ~~tipo~~ tipo

- gli argomenti formali vengono inizializzati con una copia degli argomenti attuali in modo da non modificarne il valore (passaggio per valore) ES. incremento ($\text{valore} = 6$) ($\text{valore} = 5$ non viene toccato)
- alla fine dell' elaborazione funzioni e argomenti formali vengono deallocati

4) **istante**:

- ogni volta che viene chiamata una funzione viene create una copia di funzioni e argomenti formali;
- i valori delle variabili locali non sono conservati tra le diverse chiamate

istruzione return

con l'istruzione return una funzione viene terminata (altrimenti termina a fine corso) se:

- l'espressione di return non è nulla, la funzione non produce risultato
- altrimenti restituisce il valore della funzione stessa

funzione void

funzioni che non producono risultato (ad esempio per uscite di dati)

funzioni senza argomenti

funzioni senza argomenti formali:

- possono essere chiamate "()"
- si può usare void

in uscita quando una funzione deve svolgere un blocco di codice senza utilizzare dati in ingresso

funzioni ricorsive

quando una funzione può invocare se stessa

- 1) le formulazioni prevede:
 - individuazione casi base: rappresentano le condizioni che interrompono le chiamate ricorsive, sono di essi esistono diversi all'infinito
 - definizione passi ricorsivi: ogni passo rappresenta una chiamata della funzione con parametri ridotti (che avranno le funzioni al caso base)
 - valore di ritorno: una funzione ricorsiva restituisce il valore dell'istanza delle funzioni precedente,

↓

questo valore di ritorno permette di ridurre le chiamate delle chiamate fino ad ottenere il risultato della funzione iniziale (i risultati vengono restituiti in ordine inverso)

- 2) notate bene:
 - ogni funzione può essere rappresentata in maniera iterativa che ricorsiva
 - spesso comuni ricorsive in termini di velocità e di memoria poiché non comporta le chiamate di funzioni ricorsive

↓

tuttavia se usate in maniera appropriata è più comprensibile

es

int fatt (int n)

{ if (n==0) return 1; // caso base

return n * fatt (n-1) // aggiornamento dei valori.

}

parametri come argomenti formali

usare i parametri come argomenti formali permette di riuscire a manipolare variabili esterne alla funzione

RIFERIMENTI

Un riferimento di un oggetto è un identificatore che individua l'indirizzo dell'oggetto stesso.

caso c

il nome di un oggetto è un identificatore, e contiene il suo riferimento di default
è possibile che il programmatore voglia accedere allo stesso oggetto utilizzando
degli altri di identificatore

- ↓
cioè si usa principalmente per passaggio di argomenti
come funzionalità, ciò comporta:
 - non viene creata nessuna copia
 - si lavora direttamente sul valore originale

il principio che sta alla base di ciò, è che quando muovo la funzione, non lavorerò -
più con una copia del valore ma con il suo nome (trovante è modificato)

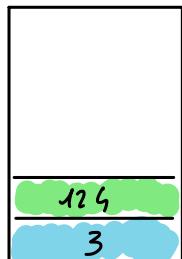
const

con il prefisso const non permette di effettuare modifiche tramite
riferimenti

Un puntatore è un tipo speciale dei fondamentali,

- è un oggetto il cui valore rappresenta l'indirizzo di un altro oggetto o di una funzione cosa succede?

indirizzo



vengono create una variabile che al suo interno memorizza il valore a cui sta puntando

120 P int
124 i int

esempio

- int main()
 - {
 - int i;
 - int *z = &i; // il puntatore ha puntato l'indirizzo di i,
 - &z = w; // ci ha scritto dentro

osservazioni:

- il puntatore va sempre initializzato, quindi per poi lavorare in questo puntatore, definirlo con $= \text{nullptr}$ // nullo

$\text{int } *z = \text{nullptr}$

- Lavoro con un puntatore, ad esempio sto sempre lavorando come puntatore, come referenze di celle di locatelli,
per differenziare e scenderci dentro: $*p = w$ // siamo dentro w

$\text{int } *p = \text{nullptr}$
 $p = p + 1$

tabelle variabili e indirizzi (memorie)

variabile	indirizzo
int i	& i
int *i	i

puntatori come argomenti formali

quando ho come argomenti formali di una funzione dei puntatori,
il momento delle chiamate deve restituire gli indirizzi delle variabili;
variabili cui sappia con che lavorare.

dentro la funzione altereremo le differenziazioni sulle modifiche dei valori.

es

```
int void (int* p, int* b)
```

*p =

*b =

```
int main () {
```

int x, y

```
void (x, y)
```

```
}
```

i puntatori si appellano di lavorare
con degli indirizzi

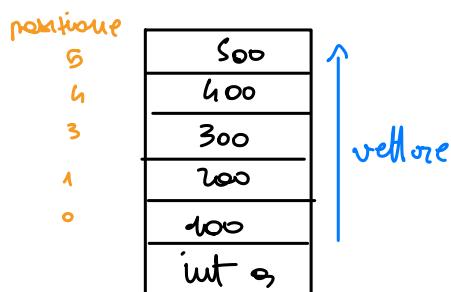
sta lavorando su x, y tramite i loro
indirizzi

ARRAY

una mappa ordinata di elementi dello stesso tipo, allo quale ci si riferisce mediante un indice che rappresenta la loro posizione nell'array

- quando definisco una variabile array, creo un vettore all'interno dello stack composto da elementi dello stesso tipo dell'array

codice e graficamente



{
int α [5] = {100, 200, 300, 400, 500}
0 1 2 3 4
cout << α [1] << endl; // 200

def.

l'identificatore dell'array identifica l'indirizzo del primo elemento dell'array

OSS: al nome del vettore è associato un valore numerico, l'indirizzo della prima componente del vettore vett = & vett

spiegazione: se punto ad esempio l'array, mi identifica la posizione 0 del vettore

utilizzo alternativo in codice

```
{  
const int N = 6  
  
int vett [N] = {0, 1, 2, 3} // il vettore deve essere riempito  
// con - 0, 1, 2, 3, 0, 0  
}
```

if numero di elementi dipende da una variabile

operazioni

non sono permesse operazioni di:

- assegnamento = $u[N] = v[N]$
- confronto = if ($u == v$) *confronto iniziativo non valori*
- aritmetica = $u + v$

funzione incrementa

```
cpp

#include <iostream>
using namespace std;

void incrementa(int v[], int n)
{
    for (int i = 0; i < n; i++)
        v[i]++;
}
```

funzione somma elementi

praticante

```
int somma( int v[], int n ) {
    int s=0
    for( i=0; i<n; i++ )
        s+= v[i];
    return s,
}

int main() {
    int v[] = { 10, 11, 12, 13, 14 } n.elementi
    cout << somma (v, n) << endl // 60
```

funzione trova massimo

```
cpp

void leggi(int v[], int n) {
    for (int i = 0; i < n; i++) {
        cout << '[' << i << "] = ";
        cin >> v[i]; stessa riga
    }
}

int massimo(const int v[], int n) {
    int m = v[0];
    for (int i = 1; i < n; i++)
        m = m >= v[i] ? m : v[i]; aggiornamento
    return m;
}

int main () {
    const int MAX = 10;
    int v[MAX], nElem; ogni linea
    cout << "Quanti elementi? ";
    cin >> nElem;
    leggi(v, nElem);
    cout << "Massimo: " << massimo(v, nElem) << endl;
    return 0;
}
```

ARRAY MULTIDIMENSIONAL

vilce generalizzata, cioè la simmetria dei vettori alle matrici con R (righe) e C (colonne) costanti.

Cookie

```

#include <iostream>
using namespace std;

int main() {
    const int R = 2; // Numero di righe
    const int C = 3; // Numero di colonne
    int m[R][C]; // Dichiara la matrice 2x3

    // Input: Lettura degli elementi della matrice
    cout << "Inserisci gli elementi della matrice" << endl;
    for (int i = 0; i < R; i++) { // Ciclo sulle righe
        for (int j = 0; j < C; j++) { // Ciclo sulle colonne
            cin >> m[i][j]; // Legge elemento (i, j)
        }
    }

    // Puntatore alla matrice
    int* p = &m[0][0]; // Puntatore al primo elemento della matrice

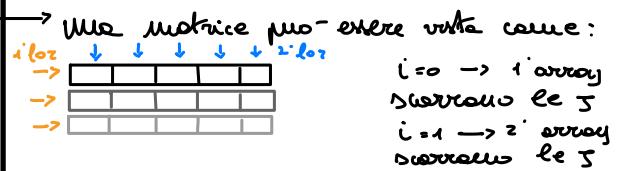
    // Output: Stampa della matrice usando il puntatore
    for (int i = 0; i < R; i++) { // Ciclo sulle righe
        for (int j = 0; j < C; j++) { // Ciclo sulle colonne
            cout << *(p + i * C + j) << '\t'; // Accesso tramite puntatore
        }
    }
    cout << endl; // Fine riga
}

return 0;
}

```

Let's analyze the code step by step:

- Input:** The code reads elements from the user and stores them in a 2x3 matrix.
- Output:** The code prints the matrix using a pointer to the first element.
- Matrix Representation:** The matrix is represented as a grid of 6 cells (2 rows, 3 columns).
- Accessing Elements:** The code uses pointer arithmetic to access elements. The expression `*(p + i * C + j)` is used to get the value at row `i` and column `j`.
- Annotations:**
 - Lettura degli elementi della matrice**: An annotation for the input loop.
 - Stampa della matrice usando il puntatore**: An annotation for the output loop.
 - Accesso tramite puntatore**: An annotation for the pointer arithmetic in the output loop.
 - riga**, **colonna**: Labels for the row and column indices `i` and `j`.
 - colonne in riga**: An annotation for the columns within a row.
 - setto a corolle per**: A note indicating that the code will be modified to use `i` and `j` instead of `i` and `col`.



inizializzatore

- int m₁[2][3] = { 1,2,3,4,5,6 } posso mettere il numero di righe [] [3] ma non di colonne, non sovrappone elementi
 - int m₂ [3][3] = { { 1,2,3 }, { 4,5,6 } } estrarre le diverse righe degli elementi in memoria

fusione sono elementi di orcey

```
cpp // viene ignorata la  
int somma(int v[4]) dimensio, in riferisce sempre a [0].  
{  
    int s = 0;  
    for (int i = 0; i < 4; i++)  
        s += v[i]; gli elementi vengono sommati ad s  
    return s;  
}
```

per gestire grandette vereibili, quindi per ulteriore
più volte la funzione:

```
cpp  
  
#include <iostream>  
using namespace std;  
  
int somma(int v[], int n) { // La funzione somma accetta un array e la sua dimensione  
    int s = 0;  
    for (int i = 0; i < n; i++) {  
        s += v[i]; // Somma ogni elemento dell'array  
    }  
    return s;  
}  
  
int main() {  
    int vett[] = {10, 11, 12, 13}; // Un array con alcuni numeri  
    int n = sizeof(vett) / sizeof(*vett); // Calcola la dimensione dell'array  
  
    cout << "La somma degli elementi e': " << somma(vett, n) << endl; // Chiama la funzione  
  
    return 0;  
}
```

STRINGS

in C++ non esiste un vero tipo stringa, esiste il concetto di catena:

catena = è simile come un array di caratteri che hanno all'interno una marca di fine stringa "tropo" per definire la dimensione dell'array

es `char cstr[] = { 'c', 'i', 'a', 'o', '\0' };`

definizioni:

- dimensione fisica = dimensione vettore, spazio occupato in memoria
- dimensione logica = dimensione del vettore precedente alla prima marca di fine stringa

inizializzazione

avviene in automatico la trasformazione:

`char cstr[] = "ciao" <=> char cstr[] = { 'c', 'i', 'a', 'o', '\0' }`

- cout << sizeof(cstr) << endl; => dimensione vettore incluso di marca d.fis.
- cout << strlen(cstr) << endl; => dimensione vettore alla prima marca d.log

informazioni in ingresso

memorizza i caratteri in ingresso e li elime nel vettore, quando trova lo spazio viene arrestata l'operazione
caratteri + 1, deve essere definita la dimensione
=> per leggere spazi = cin.getline (titolo, dimensione)

NB: se l'utente inserisce una stringa che ha un numero di caratteri maggiore della lunghezza del vettore, puo' verificarsi una corruzione della stack perché i valori da fuori campo sovrapposono a sovrscrivono ad altre variabili.

librerie <string>

1) `char* strcpy (char* dest, const char* sorg)`

mpieg. copia sorg in dest, incluso il carattere nullo e restituisce dest

cod.

```
cpp Copia codice

char* copia(char* dest, const char* sorg) {
    int i = 0;
    while (sorg[i] != '\0') { // Copia ogni carattere finché non si arriva al terminatore
        dest[i] = sorg[i]; // assegnazione contenuti di array, non gli array stessi -> i = 5 ✗
        i++;
    }
    dest[i] = '\0'; // Aggiunge il terminatore a dest
    return dest;
}
```

NB non viene verificata se la dimensione di dest riesce a contenere sorg

! il codice . `const int len2 = my_strlen (titolo)`
`char destinazione [len2+1]`
destinazione = titolo

che avrebbe il proposito di dare le dimensione di destinazione contando gli elementi da quelli di pertinenza per poi dichiarare che `dest = titolo`,
e' sbagliato, poiche' non e' possibile assegnare un array a un altro array
come comandato!

2) `char* strcat (char* dest, const char* sorg)`

mpieg. concatenare sorg al termine dest e restituisci dest (carattere nullo alla fine)

cod.

```
cpp Copia codice

char* concatena(char* dest, const char* sorg) {
    int i = 0;
    // Trova la fine di dest
    while (dest[i] != '\0') {
        i++;
    }

    int j = 0;
    // Aggiunge sorg alla fine di dest
    while (sorg[j] != '\0') {
        dest[i] = sorg[j];
        i++;
        j++;
    }

    dest[i] = '\0'; // Aggiunge il terminatore a dest
    return dest;
}
```

3) int strlen (const char* string)

spieg. calcola la lunghezza della stringa ignorando il carattere nullo

cod.

```
// Funzione personalizzata per calcolare la lunghezza di una stringa
int my_strlen(const char *string) {
    int length = 0;
    while (string[length] != '\0') { // Continua fino al carattere nullo
        length++;
    }
    return length;
}
```

4) int strcmp (const char* s1, const char* s2)

spieg. confrontare s1 con s2 :

- ritrasci un valore negativo se s1 è alfabeticamente minore (ordine dizionario) di s2
- valore nullo se sono uguali
- valore positivo se s1 > s2 (se due stringhe maiuscole e minuscole, funziona con codifica ASCII)

cod.

```
// Funzione personalizzata per confrontare due stringhe
int my_strcmp(const char *s1, const char *s2) {
    int i = 0;
    while (s1[i] != '\0' && s2[i] != '\0') { // Continua fino a che entrambe le stringhe
        if (s1[i] != s2[i]) { // Confronta i caratteri uno a uno
            return s1[i] - s2[i];
        }
        i++;
    }
    // Se le stringhe hanno Lunghezza diversa
    return s1[i] - s2[i];
}
```

5) char strchr (const char* string, char c)

spieg. scorre la stringa fino a trovare il carattere c, se lo trova restituisce il puntatore a quell'indirizzo nella stringa, se non lo trova restituisce NULL

cod.

```
// Funzione personalizzata per trovare la prima occorrenza di un carattere in una stringa
char *my_strchr(const char *string, char c) {
    int i = 0;
    while (string[i] != '\0') { // Continua fino al carattere nullo
        if (string[i] == c) { // Se trova il carattere, restituisce il puntatore
            return (char *)&string[i]; // puntatore a un carattere
        }
        i++;
    }
    return NULL; // Se il carattere non è presente, restituisce NULL
}
```

perché vengono usati i puntatori ?

il puntatore al posto di copiare il contenuto delle celle, permette di manipolare il contenuto riferendosi agli indirizzi e lavorando sulle celle originali, o addirittura scorrendo i caratteri degli array :

es : `char* p = dest
while (*p != '\0') {
 p++
}`

ORDINAMENTO VETTORE

intervento

dato un vettore avente componenti non ordinate, ottenere la versione ordinata
(crescente numerica, decrescente numerica)

ipotesi

- 1) supponiamo cosa di avere un vettore da rimaneggiare di elementi in double,
ad esempio non ci si fa a capire l'ordine
- 2) forse vorrei di avere un vettore inteso come un matrice, di lavorare però solo su
quello non servendosi di altri vettori

se non sono queste le ipotesi ci sono alternative migliori

riduzione

fare una funzione e ritornare void (precisa che solo riordinare gli elementi:
avremo un vettore in entrata e uno in uscita).

selezione - sort

- 1) ripartite del vettore unicella si visualizza il primo elemento, voglio fare in modo
che all'inizio ci sia il valore minimo di tutto il vettore
- 2) uso la funzione scambia con $i=0$ (elemento di partenza) e individuo il numero
più piccolo
- 3) individuo il secondo elemento $i=1$, individuo l'elemento minore
del sotto vettore e utilizzo la funzione scambia

N.B.: serve un ciclo del for solo per riordinare gli elementi; poiché c'è presente
un altro for dentro il for di partenza, ciò implica che dovrà di una
implementazione quadruplicata

cod.

cpp

```
// selection sort basato sulla funzione di utilita' "posMinimo"  
void selectionSort(int v[], int nElem) {  
    for (int i = 0; i < nElem; i++) {  
        int posMin = posMinimo(v, i, nElem);  
        if (posMin != i)  
            scambia(v, i, posMin);  
    }  
}
```

funzione scambia

cpp

come se fosse un puntatore

```
// scambia l'elemento i del vettore con l'elemento j  
void scambia(int v[], int i, int j) {  
    int aux = v[i]; // variabile auxiliare d'appoggio  
    v[i] = v[j];  
    v[j] = aux;  
}
```

pos minimo

cpp

```
// restituisce la posizione dell'elemento avente valore minimo,  
// fra gli elementi di v aventi indice nel range [infe, n-1]  
int posMinimo(const int v[], int infe, int n) {  
    int min = v[infe];  
    int posMin = infe;  
    for (int i = infe + 1; i < n; i++)  
        if (v[i] < min) {  
            posMin = i;  
            min = v[i];  
        }  
    return posMin;  
}
```

che problema surge?

Squando un tempo venivano utilizzate le memorie ad accesso sequenziale, l'algoritmo del selezion-sort imponeva una "scansione" per individuare il valore minimo ed inoltre per lo scambiò di valore, ed in presenza di un alto numero di valori risultava essere un problema per le componenti meccaniche.

allora

Bubble - sort

- 1) Si scorre l'array $N-1$ volte da destra a sinistra, se l'elemento da destra di pertinente è più piccolo di quello di sinistra si scambiano i valori, dopo aver completato la passata, il val. minimo è in posizione 0
- 2) Alla seconda passata, viene considerato il riconosciore non considerando $i=0$, partire da destra ed effettuare le stesse iterazioni

N.B.: al primo passaggio con elementi non ordinati (quindi se non trova elementi che sembrano) non uscirà dal ciclo e farlo terminare a differenza del selection-sort che deve fare tutte le passate

col ottimo.

c

```
void bubbleSort(int v[], int n){  
    bool ordinato = false; // per vedere se viene effettuato almeno 1 scambio  
    for (int i = n-1; i > 0 && !ordinato; i--){  
        ordinato = true;  
        for (int j = 0; j < i; j++)  
            if (v[j] > v[j+1]) {  
                scambia(v, j, j+1);  
                ordinato = false; // se questo non avviene non è true  
            }  
    }  
}
```

il ciclo si interrompe se ordinato è true +
ricorre ad ogni passaggio il più grande inoltre, si controlla ogni volta un elemento in meno
permette l'auotostoppe
ordinato = false; & se questo non avviene non è true
• ordinato = true -> si presume che sia ordinato ed ogni
 • ordinato = false -> permette nuovamente un'aulistoppe, si verifica con lo scambio

RICERCA LINEARE

interno

ricerca delle posizioni di un elemento in un sottovettore che ha come estremi il val min e val max del vettore di partenza

iterazione

- 1) scansiona tutti gli elementi effettuando confronti fra numeri adiacenti.
- 2)
 - se trova k:
 - restituisce TRUE
 - restituisce posizione k
 - se non trova k:
 - restituisce FALSE

cod.

```
c
// Ricerca Lineare: trova la posizione del primo elemento che ha valore k
// all'interno dell'intervallo [infe, supe].
// Restituisce true se l'elemento è trovato, altrimenti false.
// Se trovato, memorizza la posizione in pos.
bool ricLin(int v[], int infe, int supe, int k, int &pos) {
    bool trovato = false; // Inizialmente assumiamo che l'elemento non sia trovato
    while (!trovato) && (infe <= supe) { // Continua finché non è trovato e l'intervallo
        if (v[supe] == k) { // Controlla l'elemento all'indice supe
            pos = supe; // Memorizza la posizione se l'elemento è trovato
            trovato = true; // Aggiorna trovato per interrompere il ciclo
        }
        supe--; // Passa all'elemento precedente
    }
    return trovato; // Restituisce true se trovato, false altrimenti
}
```

RICERCA BINARIA

prerequisiti

vettore ordinato

intento

ricerca di un valore K in maniera più efficiente

iterazione

1) a partire ai parametri della funzione precedente (in cui erano presenti gli indici inf, sup) il parametro val. medio (rappresentante $(\text{inf} + \text{sup})/2$) posiziona, sui valori:

- 2) si possono verificare 3 casi:
- $K = \text{val. med.}$ → esce
 - $K < \text{val. med.}$ → ricerca K nel sotto vett. che ha sup = val. med. - 1
 - $K > \text{val. med.}$ → ricerca K nel sotto vett. che ha inf = val. med. + 1

ES: $\{ \begin{matrix} 1 & 2 & 3 & 4 & 5 \\ 1, 3, [4], [7, 8, 9] \end{matrix} \}$

3) individuato il sottovettore, viene effettuato nuovamente il processo iterativo:

- $K = \text{val. med.}$ → esce
- $K < \text{val. med.}$ → ricerca K nel sotto vett. che ha sup = val. med. - 1
- $K > \text{val. med.}$ → ricerca K nel sotto vett. che ha inf = val. med. + 1

ES: $\{ 7, [8], 9 \}$

quando questo processo iterativo termina quando $\text{val. med.} = K$

cod.

```
// Ricerca binaria: trova la posizione del primo elemento con valore k
// all'interno dell'intervallo [infe, supe] di un array ordinato.
// Restituisce true se l'elemento è trovato, altrimenti false.
// Se trovato, memorizza la posizione in pos.
bool ricBin(int ordVett[], int infe, int supe, int k, int &pos) {
    while (infe <= supe) { // Funzione nel suo insieme sup
        int medio = (infe + supe) / 2; // Calcola l'indice centrale dell'intervallo
        if (ordVett[medio] == k) { // Controlla se l'elemento centrale è k
            pos = medio; // Memorizza la posizione trovata
            return true; // Restituisce true perché k è stato trovato
        } else if (ordVett[medio] > k) {
            supe = medio - 1; // Cerca nella metà inferiore
        } else { // oppure
            infe = medio + 1; // Cerca nella metà superiore
        }
    }
    return false; // Restituisce false se l'elemento non è trovato
}
```

osservazione fondamentale

sorge spontaneo dividere, in un vettore non ordinato:

- 1) e' meglio effettuare la ricerca del valore K i.lineare
- 2) riordinare il vettore e poi effettuare la ricerca bubble-sort + i.binaria

- osservando che:

- i.lineare ha complessità-lineare
- bubble-sort + i.binaria = $O(n^2) + O(\log n) \approx O(n^2)$

vorremo sempre effettuare ricerca lineare

COMPLESSITÀ ASINTOTICA

concetto che permette di descrivere il comportamento di un algoritmo man mano che le dimensioni dell'input cresce.

- analisi qualitative della velocità esecutiva e dell'utilizzo delle memorie

notazioni O grande

è un modo formale per descrivere la complessità asintotica di un algoritmo, esprimere il comportamento peggiore di un algoritmo ignorando costanti e termini meno dominanti.

- $O(1)$ = complessità costante indipendente dalle dimensioni dell'input
- $O(n)$ = complessità lineare, tempo di esecuzione proporzionale alle dimensioni dell'input
- $O(n^2)$ = complessità quadratica, tempo di esecuzione quadratico rispetto alle dimensioni dell'input
- $O(\log n)$ = complessità logaritmica, tempo di esecuzione cresce logaritmicamente rispetto alle dimensioni dell'input

quadratica e logaritmica

1) quadratica = si verifica tipicamente in algoritmi che permutano gli elementi.

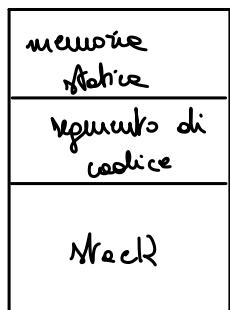
- selection / bubble sort

2) logaritmica =

- ricerca binaria

STRUTTURA RAM

partiene segmenti di una RAM (rendono access memory)



- 1) memoria statica:
 - durata = entroso per tutta l'esecuzione del programma
 - valore = parso e non oltre multi initializzatore, subisceva variazioni del codice
 - accessibilità: non eseguibile da qualsiasi funzione
- 2) segmento di codice . dove viene salvato il codice da esecuzione
- 3) stack: allocazioni variabili locali al blocco (autonodifiche) e chiamate delle funzioni
 - dinamicità: lo stack si espande e si contrae in maniera dinamica inizializzando e terminando funzioni
 - gestione LIFO (last in, first out): definisce come funzioni e variabili vengono allocate e deallocate nel mantenere un certo ordine, venne re inserita, l'ultima ad essere aggiunta e la prima ad essere tolta

STRUTTURE

una struttura e' un insieme ordinato di elementi (membri o campi) ognuno dei quali ha un nome e un tipo e contiene un'informazione

ES.

```
c  
struct persona {  
    char nome[20];  
    char cognome[20];  
    int g_nascita, m_nascita, a_nascita;  
};
```

NB: i campi della struttura non vengono intitolati: poiché riferiscono genericamente per definirli necessariamente

↓
name → variabile di tipo struct persona
struct persona p = { "Mario", "Rossi", 1, 1, 1990 }

membi di una struttura

1) un'altra struttura

ES. struct persona 1;

- 1. nome
- 1. cognome
- 1. indirizzo.via
- 1. indirizzo.città
- 1. indirizzo.cap.

ovvero ulteriori struct indirizzo

2) puntatore a struttura:

poiché "interviene" dentro una struttura, se stessa vorrebbe un ciclo infinito con occupare infatti di memoria, poniamo utilizzare un puntatore persona* proxime senza utilizzare una variabile di tipo persona:

ES struct persona 1;
struct persona 2;

persona 1. nome
persona 1. proxime = & persona 2
// collegamento a persona 2

cout << persona 1. proxime -> valore2;
// esodo tramite puntatore
// alla struttura dello stesso tipo
// al valore indicato dalle
// frecce =>

i campi di una determinata struttura si indicano con . p. nome, p. cognome.
in generale con: nome_variabile . elementi infatti:

struct persona p;

p. nome =
p. cognome =

C'è proprio come quando intitolai una variabile, solo che queste variabili hanno più "componenti"

NB. 1 = poiché le strutture si definiscono fuori dal main, è possibile definire gli elementi di una struttura anzitutto e poi nel main definire le variabili di quelle strutture (che contengono già gli elementi precedentemente definiti):

```
struct persona {  
    nome;  
    cognome;  
}
```

int main () {

persona 1, 2;

1. nome =
2. cognome =
}

operazioni su strutture

- 1) oggetti : definite
- 2) contenuto : non definite
- 3) funzioni

UNIONI

essenzialmente come le struct, riferimenti e puntatori, sono un tipo diverso e molto simili alle struct (si definiscono allo stesso modo):

- riintarsi:

```
unione numero {  
    int i;  
    double d;  
}
```

```
unione numero {  
    int i;  
    double d;  
} m;
```

numero m

- accesso:

```
m. i ;  
m. d ;
```

```
numero * mp = & m;  
mp -> i; (*mp). i;  
mp -> d; (*mp). d;
```

Differenza sostanziale tra union e struct

Una union rappresenta un'unica area di memoria che a istanti diversi contiene dati di tipo diverso:

- i veri membri di una union sono diverse interpretazioni che posso dare a quell'unica area di memoria
- occupa in memoria lo spazio relativo al membro più grande:

- in una struct {

 int i; // 4 byte +
 double d; // 8 byte = 12

- in una union {

 int i
 double d // 8 byte

→ in istanti diversi può avere ottavo o i o d
(anche se presenti entrambi)

cout << & u << endl; // 014 1d8

cout << & u. i << endl; // 014 1d8

cout << & u. d << endl; // 014 1d8

Initializzazione

union { int i; double d } m = { 4 } → a quale membro è associato? i membro
struct { int i; double d } m = { 4; 3.62 }

Per ottenere il membro, con il selettore devo definire la variabile che voglio ottenere:

u. i = 5 ottavo (se provo ad accedere all'altro mi da valori sbagliati)

PILA

Una pila è un insieme ordinato di dati dello stesso tipo in cui è possibile fare operazioni di inserimento o estrazione secondo le regole LIFO (last in, first out).

costituita

- 1) è costituita da un array
- 2) da un indice `top` che indica la posizione dell'ultimo elemento

- inserimento : `top + 1`
- estrazione : `top`

metà pila

- vuota : `top = [prima posizionale] - 1`
- piena : `top = [ultima posizionale]`

funzioni

notare che : struct pila pp { pp.stack, pp.top }

```
bool empty(const pila& pp) // pila vuota?  
{  
    if (pp.top == -1) return true;  
    return false;  
}  
  
bool full(const pila& pp) // pila piena?  
{  
    if (pp.top == DIM - 1) return true;  
    return false;  
}  
  
bool push(pila& pp, T s) // inserisce un elemento in pila  
{  
    if (full(pp)) return false; —> controllo  
    pp.stack[+(pp.top)] = s; inserimento elemento  
    return true;  
}  
  
bool pop(pila& pp, T& s) // estraie un elemento dalla pila  
{  
    if (empty(pp)) return false;  
    s = pp.stack[(pp.top)-];  
    return true;  
}  
  
void stampa(const pila& pp) // stampa gli elementi  
{  
    cout << "Elementi contenuti nella pila: " << endl;  
    for (int i = pp.top; i >= 0; i--)  
        cout << '[' << i << "] " << pp.stack[i] << endl;  
}
```

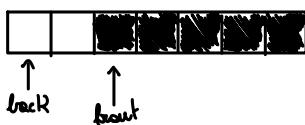
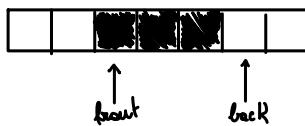
CODA

Una coda è un insieme ordinato di dati dello stesso tipo in cui è possibile fare operazioni di inserimento o extrazione secondo le regole FIFO (first in, first out).

costituite

- 1) da un array circolare (dopo l'ultima componente si ricomincia dalla prima)
- 2) da due indici:

- back = individua posizione prossima inserimento.
- front: individua posizione prossima estrazione (vuol dire la prima ad essere inserita)



stato nile

- vuota: quando $\text{back} = \text{front}$
- piena: quando $\text{front} == (\text{back} + 1) \% \text{DIM}$ cioè back è subito prima di front e non ci sono più spazi liberi, non vengono sovrascritti dati su front

Funzioni

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

typedef int T;
const int DIM = 5;

struct coda
{
    int front, back;
    T queue[DIM];
};

void inic(coda& cc) // inizializzazione della coda
{
    cc.front = cc.back = 0;
}

bool empty(const coda& cc) // coda vuota?
{
    if (cc.front == cc.back) return true;
    return false;
}

bool full(const coda& cc) // coda piena?
{
    if (cc.front == (cc.back + 1)%DIM) return true;
    return false;
}
```

```
bool insqueue(coda& cc, T s) // inserisce un elemento
{
    if (full(cc)) return false;
    cc.queue[cc.back] = s;
    cc.back = (cc.back+1)%DIM; 3%5 = (20k5) 3
    return true;                5%5 = 0; la fine
                                nella nostra
                                metraggio
}

bool esqueue(coda& cc, T& s) // estrae un elemento
{
    if (empty(cc)) return false;
    s = cc.queue[cc.front];
    cc.front = (cc.front + 1)%DIM;
    return true;
}

void stampa(const coda& cc) // stampa gli elementi
{
    for (int i = cc.front; i%DIM != cc.back; i++)
        cout << cc.queue[i%DIM] << endl;
}
```

TYPEDEF

definisce degli identificatori (nuovi tyndecl) usati per riferirsi ai tipi nelle dichiarazioni.

esempio pratica

tyndecl int T

int a = 1	T _a = 1
int b = 2	T _b = 2
int c = 3	T _c = 3
int d = 4	T _d = 4

se voglio cambiare tipo, non devo cambiargli significato ma cambio la definizione di T

DEBUGGING

processo di individuazione e correzione di errori o bug (di logica, di sintassi)

strumento

utilizzando break-point sul codice (barre verticali e orizzontali) che
stoppa il compilatore nel punto selezionato "●", successivamente è possibile
scrivere le tracce di codice uno ad uno e visualizzare nel portale
il valore corrispondente

- step in = istruzione successiva, entra nel particolare
- step over = istruzione successiva, rimani superficiale e l'istruzione seguente salta
e' possibile entrare nelle prossime istruzioni ed escludere il
codice stampa per stampa

modificare valori vettore dinamico

- devo conoscere la lunghezza del vettore
- attenzioni:
 - sul pannello di controllo segno tasto destro
 - "new watch"
 - devo { puntatore [particolare elemento], puntatore [particolare elemento] }

MEMORIA DINAMICA

nei programmi finora visti in cui venivano allocati oggetti nel main, il programmatore ha sempre dovuto specificare il numero e il tipo di tali oggetti.

così d'uso

Tuttavia non sempre si è in grado di stabilire a priori il numero di oggetti di un certo tipo che serviranno per l'esecuzione di un programma

es un array di N numeri con N dinamico e secondo delle esigenze dell'utente

memorie dinamiche

memoria globale (statica)
main.exe
heap (dinamico) (nucleo)
stack (automatica)

gli oggetti vengono qui allocati attraverso l'utilizzo degli operatori:

- new = - che ha come nome il tipo dell'oggetto da allocare e tra parentesi il valore da assegnare
- restituisc l'indirizzo ottenuto in memoria libera (dopo essersi chiesto se c'è spazio e sufficiente)
- delete = - dealloca e libera lo spazio → ha come argomento il puntatore dell'oggetto da eliminare
NB: è necessario poiché non c'è una memoria automatica in cui le variabili hanno durata finita solo quando il programma è in esecuzione

come si accede?

poiché l'oggetto allocato in memoria dinamica non ha un nome che lo identifica, e poiché non è possibile accedere a questa memoria in memoria diretta, l'oggetto allocato (indicato con new) viene associato ad un puntatore (nella rea di lavoro)

es int * pi : new int;

funzioni

- alloca vettore dinamico:

```
void alocaVettoreDinamico (double*& vett, int nite) {
```

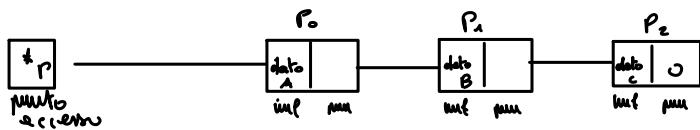
```
    vett = new double [nite]
```

NB: double*& vett sta a significare che stiamo passando per riferimento (quindi lavorare sull'originale)
un vettore che è definito con un puntatore

senza memoria dinamica non sarebbe stato possibile poiché niente è deciso nel momento

LISTE

una lista è formata da elementi dello stesso tipo collegati in una catena che varia dinamicamente, ogni elemento è una struttura (da una o più membra) contenente informazioni e da un puntatore dello stesso tipo puntante sull'elemento successivo



concetto di "dinamico corrente"

una lista deve permettere di aggiungere o eliminare le singole variabili (o vogeliini) dunque vengono allocate in memoria dinamica

allocazione

```
struct elem  
{ int inf;  
  elem* pun; };
```

funzioni

1) crea lista:

- leggere l'informazione da inserire nel vogliuccino
- allocazione nuovo elemento (vogliuccino)
- collegare il nuovo elemento al punto delle liste (da ciò si evince che i vogliacci vengono creati in testa)
- aggiornare p0 di nuovo primo elements della lista

```
lista crealista(int n)    n = elementi (vogliuccino)  
{  
    lista p0 = 0; elem* p; definisco puntatori : testa e vogliuccino  
    for (int i = 0; i < n; i++)  
    {  
        p = new elem;      p: nuova struttura, e' puntatore di accesso (elem*p)  
        elem = struct elem:{ int inf; elem* pun };  
        cin >> p->inf;    inserisco l'informazione nel campo inf  
                           (*p).int : deaferenze  
        p->pun = p0; p0 = p;  
        p0 = p;             p0 punta al nuovo  
    }                      p0 a cui da puntatore prima ; p0 punta al nuovo  
    return p0; ritorno puntatore di accesso  
}
```

2) stampa lista :

```
cpp

elem*  
void stampalista(lista p0) {  
    elem* p = p0; // creazione nuovo puntatore inizializzato alla testa  
    while (p != 0) { // finche' p(p->num) non e' zero  
        cout << p->inf << " "; // stampa informazione di p  
        p = p->pun; // aggiornamento prossimo puntatore  
    }  
}
```

ES main {

```
elem e1 h 1; nullptr}  
elem e2 h 2; nullptr}  
elem e3 h 3; nullptr}
```

e1.pun = & e2

e2.pun = & e3

elem* i = & e1 // prendi l'indirizzo del vegetuccio e1 ed assegna al puntatore i
stampalista (i)

3) dealloca lista :

```
void dealloca(elem* p0) {  
    elem* p = p0, *q; p: P0 punto dell'inizio; q: puntatore auxiliare  
    while (p != nullptr) { // scorre la lista  
        q = p; // Salva il puntatore al nodo corrente  
        p = p->pun; // Sposta il puntatore al nodo successivo  
        delete q; // Dealloca il nodo corrente  
    } // facendo così mi ottengo di non stare deallocaando  
      // prima di aver spostato il puntatore  
}
```

4) inserimento in testa:

```
elem* elem = modifica elettriva, inizializzo del p. accress  
void instesta(lista& p0, T a) Ta = (int,double...) valore  
{  
    elem* p = new elem; // 1. Allocare un nuovo elemento  
    p->inf = a; // 2. Collegare il nuovo elemento con l'informazione  
    p->pun = p0; // 3. Collegare il nuovo nodo alla lista esistente  
    p0 = p; // 4. Aggiornare il puntatore di testa della lista  
}
```

5) estrai in testa:

```
bool esttesta(lista& p0, T& a)  
{  
    elem* p = p0; // 1. Crea un puntatore a `p0`  
    // puntatore ammesso non fare delete, oltre a p0 (che viene aggiornato)  
  
    if (p0 == nullptr) // 2. Controlla se la lista è vuota  
        return false; // 3. Se la lista è vuota, restituisce false  
  
    a = p0->inf; // 4. Assegna il valore del primo nodo a `a`  
    p0 = p0->pun; // 5. Aggiorna `p0` per puntare al nodo successivo  
    delete p; // 6. Dealloca il vecchio primo nodo  
  
    return true; // operazione riuscita  
}
```

6) inserimento in fondo:

```
void insfondo(lista& p0, T a)  
{  
    elem* p; // Puntatore per scorrere la lista  
    elem* q; // Nuovo nodo da inserire  
  
    // Scorre la lista fino all'ultimo nodo  
    for (q = p0; q != 0; q = q->pun)  
        p = q; quando q è alla fine, p rimane indietro  
  
    // Crea un nuovo nodo (con valore a) e lo collega alla fine  
    q = new elem; // Alloca memoria per il nuovo nodo  
    q->inf = a; // Assegna il valore al nuovo nodo  
    q->pun = 0; // Imposta il puntatore successivo a null (fine lista)  
  
    // Se la lista è vuota, il nuovo nodo diventa il primo  
    if (p0 == 0)  
        p0 = q;  
    else  
        p->pun = q; // Altrimenti, collega l'ultimo nodo al nuovo nodo  
        // p che era rimasto all'ultimo vogliono viene collegato a q (new elem)  
}
```

7) estroi dal fondo:

```

bool estfondo(lista& p0, T& a)
{
    nullptr
    elem* p = 0;      // Puntatore per tenere traccia dell'ultimo nodo prima di quello da
    elem* q;           // Puntatore per scorrere la lista eliminare

    // Se la lista è vuota, ritorna false
    if (p0 == 0)
        return false;

    // Scorre la lista fino all'ultimo nodo (dove pun == nullptr)
    for (q = p0; q->pun != 0; q = q->pun)
        p = q;

    // Salva l'informazione del nodo finale (ultimo nodo)
    a = q->inf;

    // Controlla se il nodo finale è anche il primo nodo della lista
    if (q == p0)
        p0 = 0;          // Se la lista contiene solo un nodo, la lista diventa vuota
        eliminato l'unico vogliazione
    else
        p->pun = 0;    // Altrimenti, l'ultimo nodo viene rimosso, quindi il penultimo nod
        rimuove il collegamento dell'ultimo (mettendolo da q)

    // Dealloca il nodo finale
    delete q;

    // Restituisce true per indicare che l'operazione è riuscita
    return true;
}

```

8) inserimento in liste ordinate:

- scorrere le liste finché non si trova un vogliazione contenente un inf > e
- allora col inserire un nuovo elemento

```

void insord(lista& p0, T a)
{
    elem* p = 0; // Puntatore per tenere traccia del nodo precedente
    elem* q;       // Puntatore per scorrere la lista
    elem* r;       // Puntatore per il nuovo nodo da inserire

    // Ciclo per scorrere la lista alla ricerca della posizione giusta
    for (q = p0; q != 0 && q->inf < a; q = q->pun) q si ferma sul vogliazione > e
        p = q;           p non viene aggiornato

    // Crea un nuovo nodo e assegna il valore
    r = new elem;
    r->inf = a;
    r->pun = q; // Imposta il pun del nuovo nodo a q

    // Controlla se il nodo deve essere inserito in testa (inizio lista)
    if (q == p0) q è minore di tutti, il ciclo non soddisfa queste condizioni quindi
        non avranno ragione un iterazione
        p0 = r; // Se la posizione corretta è all'inizio della lista, aggiorna la testa
    else se non c'è così, allora continua con i passaggi
        p->pun = r; // Altrimenti, collega il nodo precedente al nuovo nodo
}

}

```

3) estrazione elemento da una lista:

```
bool estrazione(lista& p0, T a)
{
    elem* p = 0;
    elem* q;

    // Scorro la lista cercando l'elemento a
    for (q = p0; q != 0 && q->inf != a; q = q->pun)
        p = q;

    // Se q è null, l'elemento non è stato trovato
    if (q == 0) return false; caso infausto

    // Se l'elemento da rimuovere è il primo
    if (q == p0)
        p0 = q->pun; puntatore al secondo nodo
    else
        p->pun = q->pun; eliminando p punta a ciò che punta a q

    // Dealloco la memoria del nodo rimosso
    delete q;
    return true;
}
```

4) estrazione elemento lista ordinata:

```
bool estrordinata(lista& p0, T a)
{
    elem* p = 0;
    elem* q;

    // Scorro la lista finché non trovo un valore maggiore di a
    for (q = p0; q != 0 && q->inf < a; q = q->pun)
        p = q; anche >, valore aggiunto dopo

    // Se q è null o il valore che ho trovato è maggiore di a, non posso estrarre
    if ((q == 0) || (q->inf > a)) return false;

    // Se l'elemento da rimuovere è il primo
    if (q == p0)
        p0 = q->pun; il secondo nodo (cioè a cui punta q)
    else
        p->pun = q->pun;

    // Dealloco la memoria del nodo rimosso
    delete q;

    return true;
}
```

osservazione puntatore auxil

l'utilizzo di due puntatori è fondamentale quando manipoliamo strutture dati dinamiche, poiché ci permette di non perdere il riferimento a un nodo delle liste e perché ci permette di effettuare collegamenti fra essi.

- il puntatore principale scorre attraverso i nodi per trovare quello giusto da manipolare
- il puntatore auxilario memorizza il nodo precedente e supporta un'operazione come l'aggiornamento dei puntatori

cancellare da una lista tutti gli elementi aventi un valore dato

```
struct elem {  
    int val;  
    elem* pun;  
};
```

elem * p0 → [4] → [4] → [5] → [3] → [4] voglio eliminare gli elementi
che hanno 4

for

- 1) controllo a partire dalla testa tutti gli elementi: se l'intero è 4
- 2) gestisco se ho dunque un elemento con intero ≠ 4

codice

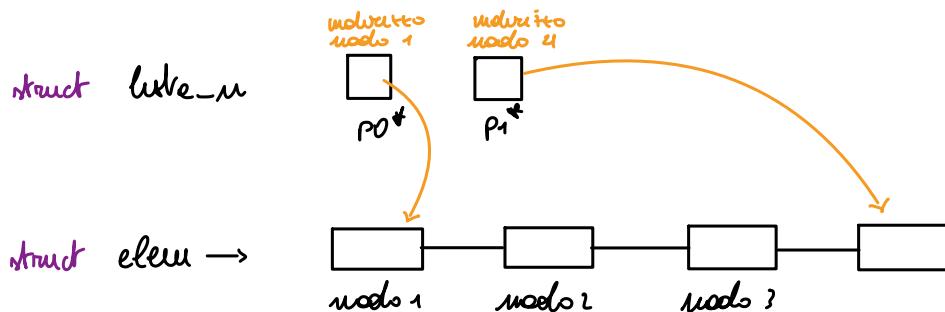
liste con puntatore coda

È possibile gestire una lista con più di un puntatore a lista, ad esempio con un puntatore p_0 alla testa e un puntatore p_1 alla coda, ciò facilita notevolmente le operazioni su una lista:

```
struct lista_n {  
    elem* p0;  
    elem* p1;
```

In questo caso, questa struttura è un contenitore di due puntatori che riguardano i valori degli membri di determinati oggetti.

Spiegazione grafica



Funzioni

1) creare lista:

```
lista_n crealistai(int n)  n = numero di nodi  
{  
    elem* p;           // Puntatore temporaneo per creare nuovi nodi di lista  
    lista_n li = {0, 0}; // Inizializzazione della lista: p0 e p1 inizialmente nulli (0)  
  
    if (n >= 1)          // Se ci sono nodi da creare  
    {  
        p = new elem;   // Creiamo il primo nodo  
        cin >> p->inf; // Inseriamo un valore nel campo `inf` del nodo  
        p->pun = 0;      // L'ultimo nodo non punta a niente (pun = nullptr)  
        li.p0 = p;       // Il primo nodo è sia il primo che l'ultimo nodo  
        li.p1 = p;  
  
        for (int i = 2; i <= n; i++) // Creiamo i successivi nodi  
        {  
            p = new elem; // Creiamo un nuovo nodo  
            cin >> p->inf; // Inseriamo il valore nel campo `inf`  
            p->pun = li.p0; // Il nuovo nodo punta al nodo attualmente puntato da `li.p0`  
            li.p0 = p;       // Aggiorniamo `li.p0` per puntare al nuovo nodo  
        }  
    }  
    return li;           // Restituiamo la lista  
}
```

2) estrarre in testa:

```
bool esttesta1(lista_n& li, T& a)
{
    elem* p = li.p0; puntatore auxiliario
    if (li.p0 == 0)
        return false;
    a = li.p0->inf; valore il valore
    li.p0 = li.p0->pun; il valore p0, nuovo: vorrebbe eliminato l'oggetto cui punta,
    il puntatore p0 è ora diventato il corrispondente al nuovo nodo
    delete p;
    if (li.p0 == 0) c'era solo un nodo nella lista, che ora è stato eliminato
        li.p1 = 0;
    return true;
}
```

3) inserire in fondo:

```
void insfondo1(lista_n& li, T a)
{
    elem* p = new elem; // 1. Crea un nuovo nodo
    p->inf = a; // 2. Assegna il valore 'a' al campo 'inf' del nuovo nodo
    p->pun = 0; // 3. Imposta il puntatore 'pun' del nuovo nodo a nullptr

    if (li.p0 == 0) // 4. Verifica se la lista è vuota
    {
        li.p0 = p; // 5. Se la lista è vuota, il nuovo nodo è Testa e coda
        li.p1 = p;
    }
    else
    {
        li.p1->pun = p; // 6. Se la lista non è vuota, aggiorna puntatore ultimo nodo
        li.p1 = p; // 7. Imposta il puntatore 'p1' della lista al nuovo nodo
    }
}
```

! N.B.: tutte le funzioni precedentemente viste possono essere utilizzate se ricondotte sulle con liste a due puntatori.

- Leggi vettore dinamico da tastiera:

void leggiVettoreDinamicoDaTastiera (double *q, int lun) {

```
for (int i=0; i<lun; i++)
    cin >> q[i]
}
```

NB: vengono inseriti i valori nel vettore

- Estendi vettore dinamico:

void estendivettoreDinamico (double *&vd, int lun, int a) {

double *vettoreEsteso = new double [lun+1]; creazione

```
for (int i=0; i<lun; i++)
    vettoreEsteso[i] = vd[i]; come dei dati.
```

vettoreEsteso[lun] = a;

delete [] vd; cancella vecchio

vd = VettoreEsteso; modifica puntatori.

}

È molto costosa l'estensione di un vettore dinamico

- dealloca vettore dinamico:

void deallocaVettoreDinamico (double *&p) {

delete [] p;

p = null ptr

matrice in memoria dinamica

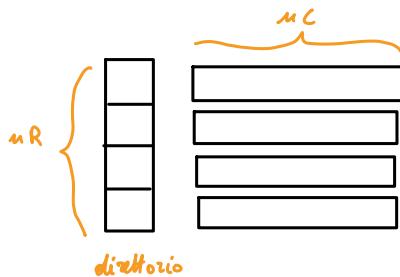
per allocare una matrice in memoria dinamica potremo intraprendere due strade:

1) int main() {

```
    cin >> nR  
    cin >> nC  
    int * p = new int [nR * nC]  
}
```

tuttavia, poiché in realtà stiamo creando un unico vettore, sarà più difficile lavorarci sopra.

- 2) (più conveniente) - dobbiamo prima allocare un vettore colonna per tanti elementi:
quegli sono i numeri di righe (direttorio)
- poi bisognerà allocare all'interno di quei vettori righe separate con le colonne



- codice:

```
int *** direttorio = new int ** [nR];  
|| * direttorio = new int * [nC] // allocazione prima riga  
| * direttorio[2] = new int [nC]  
  
dimensionalmente for (int i=0; i<nR; i++)  
    *direttorio[i] = new int [nC]
```

- letture.

```
for (int i=0; i<nR; i++) {  
    for (int j=0; j<nC; j++)  
        cout << p[i][j] << endl;  
}
```

VISIBILITÀ

programmi semplici o complessi

- 1) semplici:
 - contenuti in un solo file
 - poche funzioni che riscontrano informazioni con argomenti e risultati.
- 2) complessi:
 - tecniche di programmazione modulare:
 - il programma è suddiviso in moduli (ognuno dei quali svolge un compito)
 - i moduli si scambiano informazioni utilizzando oggetti comuni
 - **es.** programma gestione vendite di un negozio, moduli per calcolo vendite, gestione inventario, gestione pagamenti

concetto di visibilità

il concetto di visibilità si riferisce a dove una variabile, funzione o qualunque identificatore è accessibile all'interno di un programma

- visibilità globale: variabile o funzione accessibile in qualsiasi parte del codice
- visibilità locale: variabile dichiarata all'interno di una funzione è visibile solo all'interno di quelle funzione

regole di visibilità

servono a controllare la condivisione delle informazioni fra i vari componenti di un programma:

- permettono a più parti del programma di riferirsi ad una stessa entità per poter manipolare le informazioni
- impediscono ad alcune parti di un programma di riferirsi ad un'entità estendo modifiche o manipolazioni

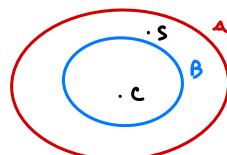
BLOCCI

un blocco e' costituito da una sequenza di istruzioni racchiusa fra {}.

↳ Un'istruzione puo' essere costituita da un blocco, per cui si possono avere blocchi annidati.



un identificatore dichiarato in un blocco e' locale al blocco stesso



s = variabile in A e B
c = variabile in B

N.B.: se B definisce s = ugualare il soprastante

UNITA' DI COMPILAZIONE

Una unita' di compilazione e' costituita da un file sorgente

SPAZIO DI NOMI

insieme di dichiarazioni e definizioni racchiuse tra parentesi graffe, ognuna delle quali introduce determinate entità dette membri.

Un membro può avere un altro spazio di nomi, perciò è possibile l'annidamento.

namespace uno

```
{ struct st1 { int a, double d };  
    int n;  
    void ff (int a)  
}
```

namespace due

```
{ struct st2 { int a, double d };  
}
```

main () {

- uno::st1 ss1,

sto verando ss1 che c'è una struct del libro uno

- using namespace due;

ss2 ss2;

- gli identificatori relativi ad uno spazio di nomi sono visibili dal punto in cui sono dichiarati fino alla fine dello spazio dei nomi
- al di fuori di uno spazio di nomi, gli identificatori relativi ai suoi membri possono essere resi visibili mediante l'operatore di visibilità `::`



se viene usato spesso, è possibile usare la direttiva `using namespace`

problematiche `using namespace`

a volte la direttiva `using namespace` può occorrere ambiguità:

namespace aaa

```
{ int n;  
}
```

namespace bbb

```
{ int n;  
}
```

main () {

`using namespace aaa;`

`using namespace bbb;`

`n=10 // ambiguità`

`aaa::n = 10`

proprietà - nome spazio

uno spazio di nomi si definisce aperto, nel senso che è possibile utilizzarne più volte lo stesso identificatore di spazio di nomi in successive dichiarazioni per richiedere ogni volta nuovi membri

nuovo spazio uno

{ int n;

}

il nuovo spazio uno è { int n;

int m,

}

nuovo spazio uno

{ int m;

}

livello globale

In C++ esiste per default uno spazio di nomi globale definito a "livello di file": i membri di tale spazio di nomi si riferiscono con l'operatore di risoluzione di visibilità ::

int i = 4 } livello globale
int j = 5

main () {

int l; // 4

{

int i = 2 visible al blocco

cout << i << ::l << endl; // 2, 4

}

}

COLLEGAMENTO

un programma può essere formato da più unità di compilazione, che vengono sviluppate separatamente e poi collegate per formare i file eseguibili: una è una sola deve contenere la definizione della funzione main()

concetto di collegamento o legame

un identificatore ha . • collegamento interno se: - è visibile solo all'interno della stessa unità di compilazione
- non può essere usato nelle altre
 es: static int counter = 0
• collegamento esterno se: - è visibile anche in altre unità di compilazione
- dichiarato con **extern** e definito nella sua unità

regole default

- 1) gli identificatori di entità definite a livello di blocco hanno collegamento interno
- 2) gli identificatori di entità definite a livello di file hanno collegamento esterno
(a meno che non siano definiti con **const**)

collegamento esterno

come già ripetuto, oggetti e funzioni possono essere utilizzati in altre unità di compilazione, in ogni altra unità dovranno essere dichiarate (ma non definite, altrimenti verrà allocato un nuovo spazio in memoria e si genererebbe un errore di multiple definizioni)

- oggetto: - viene dichiarato con **extern**
- viene definito se non si usa **extern** e viene attribuito un valore
- funzione: - viene solo dichiarata se si specifica l'interfaccia
- viene anche definita se si specifica il corpo

CLASSI DI MEMORIZZAZIONE

La classe di memorizzazione (storage class) di un oggetto è una proprietà che riguarda il suo tempo di vita, cioè quanto viene mantenuto in memoria

- **automatica**: per default sono quelli definiti: in un blocco, hanno durata di vita pari alla durata del blocco, la variabile viene initializzata con lo stesso valore ogni volta che viene invocata
- **statica**: per default sono quelli definiti al di fuori delle funzioni, hanno durata di vita pari alla durata del programma.

osservazione automatica

Tuttavia, se vogliamo far rimanere invariato un valore assegnato di una variabile di blocco (automatica) permanente anche **statico**, il campo di visibilità resta comunque limitato, però viene creata in memoria alle prime istruzioni e la memoria ad essa assegnata viene mantenuta fino le istruzioni successive (quindi il valore non si esalta col ogni invocazione)

uso di static

- 1) in una definizione di oggetti o di funzioni a livello di file, indica un collegamento interno
- 2) in una definizione di oggetti a livello di blocco, rende "statici" gli oggetti stessa

EFFETTI COLLATERALI

per effetto collaterale si intende il processo con cui delle funzioni modificano il valore di variabili visibili al di fuori del suo ambito locale. (passaggio per riferimento o puntatori)

vantaggi e svantaggi

1) vantaggi: sicuramente è meglio utilizzare funzioni pure (passaggio per valore) poiché altrimenti le f. con effetti collaterali renderebbero il flusso di informazioni più costoso rendendosi più esposti ad errori.

NB: nel caso di più file, se un valore viene modificato attraverso le funzioni si potrebbe non capire da dove venga fuori.

2) vantaggi: tuttavia a volte si elabora un quantità di informazioni molto grande da la loro copiatura riunite nelle costanti

NB: le scelte tra riferimento e puntatori si basa su considerazioni di leggibilità, i puntatori possono mettere in evidenza le chiamate di funzioni con effetti collaterali (che fanno modifiche).

MODULI

Un modulo è una parte di programma (costituita da uno o più file) che offre una data funzionalità.

tipi di moduli:

l'interazione fra moduli può essere vista in termini di offerte e uso di servizi:

- servizi = risorse (variabili, funzioni...) → interfacce tra moduli
- modulo servitore = offre le risorse
- modulo cliente = che le usa

NB: uno stesso modulo può avere sia servizi che clienti

modulo servitore

è in genere costituito da due file:
• interfaccia = contiene le dichiarazioni dei servizi;
file di tipo .h
• realizzazione = contiene le definizioni:
- delle varie dell'interfaccia
- di entità usate nel modulo

interfaccia e realizzazione

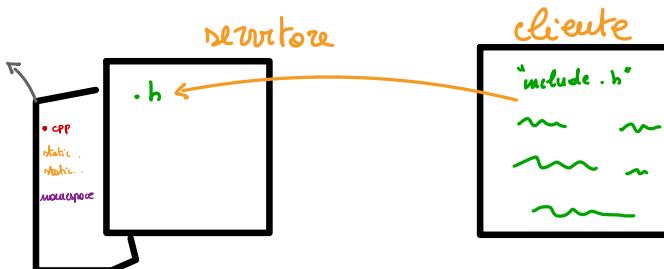
- 1) interfaccia:
 - è la faccia del modulo,cio- che il modulo offre,
 - contiene le dichiarazioni ma non spiega come funzionano
- 2) realizzazione:
 - è il dietro le quinte
 - contiene la logica delle funzioni e variabili

Il cliente vede solo l'interfaccia .h e non i dettagli di come funziona (.cpp),cio- permette:

- nascondere i dettagli di implementazione
- permette di modificare la realizzazione di un modulo senza influenzare il funzionamento dei clienti (la firma non include ancora la stessa)
- semplificare le dipendenze fra moduli (il cliente usa le varie senza dipendere dall'implementazione interna, poiché è nascosta)

NB: il principio della separazione tra interfaccia e realizzazione di un modulo servitore, e dall'indipendenza dei moduli servitori dai clienti, è noto come ocultamento dell'informazione

nel .cpp, include .h public
se devi cambiare ecc.
voglio includere:



modulo cliente

dunque il modulo cliente:

- importa ed utilizza le risorse offerte dal modulo servitore
(include i file di interface **.h**)
- non ha bisogno di conoscere i dettagli di implementazione
del modulo servitore. **.cpp**)

astrazioni procedurali

1) entro dei moduli servizi che realizzano le cosiddette astrazioni procedurali:

- mettono a disposizione degli altri moduli un insieme di funzioni attraverso l'occultamento dell'informazione: - le definizioni di queste funzioni si trovano in un file di interfaccia incluso dai moduli clienti.
↓
queste funzioni riservano metà spazio da non necessarie danno conoscenze della loro struttura interno

2) alcuni moduli servizi gestiscono una struttura dati, accessibili al cliente soltanto attraverso funzioni e non direttamente alla realizzazione interna => in questo modo si ottengono oggetti astratti

es: file u-pila → realizza un oggetto astratto poiché gestisce una struttura pp { int top
variabile solo all'interno del modulo
int stack } }

u-pila.h u-pila.cpp

N.B.: un oggetto astratto si riferisce ad un'entità che è definita in base alle sue caratteristiche visibili cioè esterne (interfaccia) e non delle sue strutture interne (implementazione)

limitazioni dei tipi derivati

un tipo di dato astratto dunque è costituito da un insieme di valori e da una serie di funzioni definite su di esso dello stesso tipo dei valori. Possono essere definite operazioni oggettive che non necessario far uso delle rappresentazioni interne

astrazioni

I moduli servitori possono avere ulteriori in varie tipologie di astrazione

1) astrazioni procedurali: alcuni moduli mettono a disposizione di altri moduli un insieme di funzioni secondo il principio dell'ocultamento,
nella rete tali funzioni vengono usate senza avere conoscenza della struttura interna **es:** librerie <string.h>

2) oggetti astratti: alcuni moduli servitori forniscono una struttura dati, la struttura interna non è accessibile al modulo cliente, il quale può accedere tramite funzioni dichiarate nell'intestazione.

NB: oggetto astratto, poiché è un oggetto (ad esempio di una struttura) che viene manipolato attraverso funzionali

3) tipi di dato astratti: alcuni moduli servitori forniscono un tipo di dato (primitivi, derivati), secondo il principio dell'ocultamento la struttura interna non è accessibile ed i moduli clienti possono accedere solo tramite le funzionali di interfacce

Tuttorie, no! non basta

le regole di visibilità e collegamento non bastano per ottenere un tipo di dato astratto:

- prendiamo ad esempio la pila:

anche se il file di intestazione pila.h espone solo le funzioni
il modulo cliente ha accesso diretto ai membri pubblici di pila {top, dati}
poiché deve poter creare oggetti di tipo pila

limitazioni tipi derivati

il concetto di tipo di dato astratto viene realizzato completamente dai tipi fondamentali a differenza di quelli derivati:

tipi fondamentali

```
int min(1);  
int x=10;  
int y=20;  
  
int somma = x+y // 30  
int differente = y-x // 10
```

tipi derivati

ottenendo tipi derivati, vengono applicate anche operazioni per accedere alle componenti:
anche se si utilizza la programmazione modulare

i tipi di dato astratti vengono realizzati completamente con le classi, che estendono il concetto di struttura

NB non mi accade mai di dettagli interno (rappresentazione in memoria)

CLASSI

Un tipo classe ha una parte:

- **privata** = contiene la struttura dati che realizza il tipo (gli attributi)
- **pubblica** = contiene le dichiarazioni delle funzioni proprie del tipo e costituiscono l'interfaccia (**funzioni membro**) e sono le uniche che possono accedere alle parti private

dichiarazione

La dichiarazione di una classe è simile a quella di una struttura, ma le sezioni (composte da membri) possono essere diverse in relazione private e pubbliche grazie agli indicatori accesso.

class nome

```
{ parte privata      // se non viene specificato l'indicatore è privata  
protected :  
parte protetta  
public :  
parte pubblica  
};
```

membri

Un membro di una classe può essere:

- un tipo (enum, struttura)
- un insieme di dati (int) (operatore)
- una funzione
- una classe (dicona della classe di appartenenza)

dichiarazione funzioni

- le funzioni membri della dichiarazione sono definite inline. Il compilatore sostituisce la chiamata con il corpo delle funzioni
- le funzioni membri possono essere dichiarate anche esternamente utilizzando l'operatore di risoluzione di visibilità ::

es: class complesso

{ parte privata

public :

parte pubblica.

5

void complesso :: init (... , ...)

OGGETTI CLASSE

Un oggetto appartenente a una classe si dice oggetto classe o classe e negli ha un tipo classe-type-identificatore

Oggetto

Proprio come le struct, per lavorare su un oggetto dovre negli mettere **nuovo oggetto**. funzione()

Copia

un oggetto può essere inizializzato con gli stessi valori di un oggetto già esistente:

```
class complesso x,y  
{  
    ...  
}
```

- complesso s = x; // copiatura valori
- complesso s(x);

Operazioni

Le operazioni sugli oggetti di una classe si dividono in due categorie:

1) Operazioni proprie delle classi:

Rappresentate dalle funzioni membro dichiarate nella parte pubblica, e sono invocabili tramite:

- oggetto = object.function()

ES: class persona P;
P. salute();

- puntatore = pointer \rightarrow function()

ES: persona * p = new persona
p \rightarrow salute();

2) Operazioni predefinite: sono quelle che il compilatore fornisce per gli oggetti di una classe:

- copiatura
- inizializzazione
- **NO** (confronto)

Passaggi e funzioni

Gli oggetti classe possono essere passati a funzioni per valore o per riferimento

PUNTORE THIS

è un puntatore che fa riferimento all'oggetto su cui è invocate le funzioni

Principio

quando una funzione membro viene chiamata su un oggetto, **this** rappresenta l'indirizzo dell'oggetto che ha invocato la funzione

- **this** permette di riferirsi esplicitamente ai membri dell'oggetto

es: class complesso
{ parte privata }
public:
double reale();
};

double complesso:: reale() {
return **this** -> re; } ; // il reale di questo oggetto classe

- 1) virgole nelle ✓
- 2) escl. ✓
- 3) i belli faccioni ✓
- 4) numeri array ✓

visibilità a livello di classe

- gli identificatori: dichiarati all'interno di una classe sono visibili dal punto della loro dichiarazione fino alla fine della classe (quindi se dichiari una funzione privata dopo una variabile, questa non verrà - vista)
- funzioni membro: seguono tutte quelle se fattesse dichiarate alla fine, quindi possono accedere a tutte le variabili della classe
- riserva identificatore: se all'interno di una classe viene definita una variabile già dichiarata all'esterno della classe, quest'ultima viene sovrascritta da quelle più interne
- operatori di selezione: al di fuori delle classi si può accedere ai membri attraverso l'operatore :::
 - funzione membro => class myclass {
public:
 void myfunction();
};
void myclass::myfunction()
{
};
 - tipo enum => class myclass {
public:
 enum color {RED, GREEN, BLUE};
};
Myclass::color c = Myclass::RED
interna *lo comincia da re*
 - membri statici => quando dichiara static int i; e noi faccio Myclass::static int i = 42, sto assegnando 42 a tutte le istanze di una classe che contengono static int i
↳ * non sono statici possono lavorare su istanze di classi

N.B: nelle classi annidate, l'istanza è locale a quella esterna bisogna usare due operatori di risoluzione di visibilità.

MODULARITÀ E RICOMPOSIZIONE

l'uso di classi per la creazione di tipi di dati astratti permette di scrivere programmi in cui l'interazione fra moduli è limitata alle interfacce (parte pubblica) e non alla implementazione o alla parte privata.

riparazione

Le diverse in parte pubbliche e private non è sufficiente per una gestione ottimale, questo perché quando dobbiamo modificare la classe, dobbiamo ricompilare anche i moduli che utilizzano l'interfaccia di quella classe.

Perciò si lavora con file .h e .cpp precedentemente illustrati.

FUNZIONI GLOBALI

Le funzioni globali sono quelle che non appartengono a classi (tra cui le funzioni main), tuttavia possono operare con le funzioni membri dichiarate nella parte pubblica.

costruttori

d'oggetto classe può essere inizializzato per mezzo di una funzione membro, negli esempi visti finora avevamo proprio una funzione "inizializza ()"

Tuttavia un oggetto classe si può inizializzare contestualmente alle sue definizioni mediante un costruttore (funzione costruttore che ha il nome della classe)

prima

```
class eleenco  
{ int re; int }  
nullis;  
    iniz();  
/  
int main () {  
eleenco f,  
f. iniz();
```

- 1) creazione infallita
- 2) inizializzazione

dopo

```
class eleenco  
{ int re; int }  
nullis;  
    eleenco (int l, s);  
  
eleenco:: eleenco (int l, s) → la ist defn cod.  
{ re=l; im=s }  
(va bene anche in un file .cpp)  
  
int main () {  
complexo x (3,7)  
    ↓  
creazione oggetto "x"
```

N.B.: per una stessa classe possono essere definiti anche più costruttori

costruttori default

- esistenziale: con un costruttore è obbligatorio fornire gli argomenti per la funzione e non c'è più possibilità di non inizializzare
- produttive:
 - meccanismo overloading: - viene creato un nuovo costruttore senza argomenti e con parametri = 0 che riceverà l'inizializzazione nel corso in cui non vengono forniti tutti gli argomenti:

ES: complexo (int i; int k,);
complexo ();

complexo:: complexo (int i; int k,);
{ re=i; im=k }
complexo:: complexo ();
{ re=0; im=0 }
elementi della classe

int main () {
complexo c1 (3,4);

complexo c2; OK
complexo c3 (3) NO
// deve fornire due argomenti

- meccanismo argomenti default. al posto di complexo () metto complexo (int i=0, int k=0)
così poi dichiaro complexo:: complexo (int i; int k) { int re=re; int im=im }
e posso inserire 2, 1 o 0 valori:

```
complexo c1 (3;4); // (3,4)  
complexo c2 (3); (3)  
complexo c3; (0,0)
```

Allocazione in memoria libera

Supponiamo un costruttore che richiede l'allocazione in memoria libera, come ad esempio per le stringhe.

```
class strunge  
{ char * str; }  
public:  
strunge ( const char s [] );
```

```
strunge :: strunge ( const char s [] ) {  
    str = new char [ strlen ( s ) + 1 ]
```

```
int main () {  
    strunge str (" fondamenti di programmazione " )
```

- In generale: new plurale^* $\text{pc} = \text{new} \text{ complesso}$ (3,4)

Regole di dinamica

I costruttori di oggetti che creano riferimenti usano le seguenti regole:

- 1) per gli oggetti **statici**, all'inizio del programma (prima delle funzioni **main**)
- 2) per gli oggetti **autonomi**, (con funzioni) quando viene incontrata la loro definizione (funzione costruttore)
- 3) per gli oggetti **dinamici**, quando viene incontrato l'operatore **new**
- 4) per gli oggetti **memoriai** di altri oggetti, quando quest'ultimo raggiunge i valori nulli:
(un oggetto memoriai di un altro oggetto, viene creato prima di quest'ultimo al momento della dinamica)

DISTRUTTORI

poiché, come abbiamo già ripetuto più volte, la memoria dinamica non si dealloca da sola, è necessario introdurre un distruttore.

funzione

~ maneggiato () ;

per�icolante

- viene invocata automaticamente (a meno che non si tratti di memoria dinamica)
- non ha argomenti
- ha struttura: `Strange::~Strange { delete [] pun };`

regole di chiamate

- 1) oggetti statici : al termine del programma
- 2) oggetti esistenti : all'urto del blocco
- 3) oggetti dinamici : quando incontra l'operatore `delete`

COSTRUTTORE COPIA

il costruttore di copia è un particolare costruttore che opera fra due oggetti effettuando una ricopistatura membro a membro dei campi dati, viene applicato:

- 1) un oggetto dove viene initializzato con un altro oggetto della stessa classe
- 2) un oggetto dove viene passato ad una funzione come argomento valore
- 3) una funzione restituisce come valore un oggetto classe (con return)

(esempi)

1) complesso c1 (3,4) // initializzatore

compleksso c2 = c1 // copia

2) compleksso (compleksso& c) { re=c.re; im=c.im }

quando chiamo la funzione viene creata una copia di c1 facile alle mutazioni

3) compleksso creacompleksso() { compleksso temp(1,2); return temp } compleksso c = creacompleksso()

memorie dinamica

Se il costruttore copie non viene redenituito, viene usato un costruttore copia predefinito che copia i membri uno ad uno, quindi anche i puntatori, creano due puntatori che puntano allo stesso oggetto

* possiamo inserire la funzione copia nella parte privata per rendere la classe inaccettabile: copia (copia &) in privato

corretto costruttore copia

```
class Informazioni {
    char* dati; // Memoria dinamica per i dati
public:
    // Costruttore parametrico
    Informazioni(const char* str) {
        dati = new char[strlen(str) + 1];
        strcpy(dati, str);
    }

    // Costruttore di copia (Deep Copy)
    Informazioni(const Informazioni& orig) {
        dati = new char[strlen(orig.dati) + 1]; // Alloca nuova memoria
        strcpy(dati, orig.dati); // Copia il contenuto
    }

    // Distruttore
    ~Informazioni() {
        delete[] dati; // Libera la memoria dinamica
    }
}
```

FUNZIONE FRIEND

Una funzione friend viene introdotta con la parola chiave **friend**, è una funzione che può non fare parte della classe, ma il suo nome di esecuzione si inserisce privato e protetto di quella classe (non solo coi pubblici)

ESEMPIO

```
class MyClass {  
private:  
    int x; // membro privato  
  
public:  
    MyClass(int val) : x(val) {} // costruttore  
  
    // Dichiarazione di una funzione amica  
    friend void printX(const MyClass& obj);  
};  
  
// Definizione della funzione amica  
void printX(const MyClass& obj){  
    std::cout << "x = " << obj.x << std::endl; // Accede al membro privato 'x'  
}
```

Attenzione tutti i valori di un oggetto classe ed una funzione, quest'ultima ha accesso ai membri interni dell'oggetto

OSSERVAZIONE IMPORTANTE classi concorrenti

1) supponiamo di avere una classe A: class A {
 private:
 public:
 class B
 }

In questo caso la classe B ha accesso solo al pubblico di A e al privato di B (ne connaît la structure classe)

```
class A {  
private:  
public:  
    friend class B  
}
```

In questo caso class B ha accesso anche ai membri di A

2) è possibile, a partire da due classi, definire una sola funzione di class B come friend di class A, secondo l'operatore di risoluzione di visibilità

OVERRIDING DI OPERATORI

Un tipo di dato estratto può richiedere che vengano definite alcune operazioni: è opportuno che vengano rappresentate con gli stessi operatori analoghi dei tipi fondamentali, piuttosto che con funzioni.

overlooking

il principio dell'overloading, oltre che per le funzioni con lo stesso nome (il caso dei costruttori), può essere applicato anche per gli operatori.

Definizione

si fa attraverso l'uso di operatori seguiti dall'operatore da definire

- ogni volta che viene usato nell'oggetto classe equivale a una chiamata alla funzione operator

Moro / Binario

1) un operatore unario:

- opera su un solo operando
 - della forme $(++, --, +, -)$
 - implementati con:

- une fonction membre : non indépendante (génére l'ensemble du répertoire)

numbers operator - () return numero(-valore) // regno opposto

- une feuille globale : réunit un argumento

numbers operator - (numero & m) f' return numbers (-m. valore)

2) operatore binario:

- opera su due operandi
 - del tipo: ($+$, $-$, \cdot , \backslash ...)
 - implementati con:

- funzione membro: riceve come argomenti l'altro operando

intake 1 (concrete)

intente 2

valore corrente (.this)

numero aerotext (numero & offerte) / return numero (valore & offerte valore)

- huitième globule: récapitulation

numbers operator + (numbers & n1, numbers & n2) {
 return numbers (n1.value, n2.value);

operatori che si possono ridefinire

Si possono ridefinire tutti gli operatori tranne:

- L'operatore risoluzione di visibilità (::)
- L'operatore selezione di membro (.)
- L'operatore selezione di membro attraverso un puntatore a membro (.*)

ATTENZIONE: gli operatori di assegnamento ('='), di indicizzazione ('[]'), di chiamata di funzione ('()') e di selezione di membro tramite un puntatore ('->*') devono essere ridefiniti sempre come funzioni membro.

ATTENZIONE: oltre all'operatore di assegnamento, sono predefiniti anche quello di indirizzo ('&') e di sequenza (';').

operatori incremento

per le sintassi precedente vale:

complesso & `operator++()` { `z++;` `inc++;` `return *this` }

ma non è
il valore incrementato

`c1++;` } il valore incrementato

osservazioni

- 1) gli operatori di incremento: si definiscono come funzioni membri e restituiscono un riferimento all'oggetto
- 2) il compilatore considera le convenzioni di tipo, se entro più possibilità valde viene generato un errore

NB: non assegnare un significato non intuitivo agli operatori

OSS: le versioni predefinite degli operatori sostengono diverse equivalenti: $x++ \Leftrightarrow x+=1$
al momento dell'overloading queste equivalenti vengono rispettate

restrizioni sull'overloading

- 1) si possono ridefinire solo operatori già esistenti, ma non si possono creare altri
- 2) non si possono cambiare le proprietà (+ deve essere associativo e commutativo)
- 3) almeno 1 argomento deve essere di un tipo dell'utente

es. int operator+(int a, int b) No!

OPERAZIONI LETTURA/SCRITTURA

In C++ uno stream corrisponde a un'infante della classe, le cui:

- parte pubblica: contiene le operazioni disponibili
- parte privata: contiene le strutture dati, tra cui un buffer che memorizza un blocco di caratteri in transito fra la memoria centrale e il dispositivo ingresso/uscita

overloading

tutte queste classi di ingresso e uscita utilizzano il meccanismo dell'overloading per definire le operazioni di lettura e scrittura dei diversi tipi di dato

tipi di operazione

- formattate: il contenuto delle caselle viene interpretato come codice di caratteri, attraverso una conversione in dati interi in base 2 o in esadecimale, e viceversa per la scrittura
- non formattate: non viene effettuata alcuna trasformazione, ma vengono trasferite le sequenze di byte stesse \rightarrow memoria

accesso

- sequenziale: accesso secondo un ordine arbitrario (file di testo, input da tastiera)
- casuale: accesso diretto a qualsiasi posizione, richiede funzioni di puntamento (SSD, database)

ISTREAM / OSTREAM

nelle librerie di ingresso e uscita sono definite le classi istream per l'ingresso, ostream per l'uscita.

- iostream: infante della classe istream
- cout, cerr: infante della classe ostream

ISTREAM

```
class istream
{ // stato: configurazione di bit
public:
    istream(){...}; portogallo per informarre, permette variazione
    migrazione fra () quando viene usato
    istream& operator>>(int&); legge caratteri e li trasforma in dati interi
    istream& operator>>(double&); il nuovo caratteri sono in memoria delimitati
    istream& operator>>(char&); // (1)
    istream& operator>>(char*); // (2)
    istream& operator>>(char* buf, int dim, char delim = '\n'); // (3)
    istream& read(char* s, int n); // (4)
    ... legge n byte e li memorizza a mettere dato nel punto di 3
};
```

introduzione

basic - input - expression - statement

input - stream >> variable - nome;

array

- rilevare dello stream di una sequenza di caratteri, perché seguono le regole specifiche per quel tipo di dati
- convertere di queste sequenze in un valore assegnato ad una variabile

esecuzione del programma lettura (char, int, double, string)

quando viene incontrata un'istruzione di lettura:

- il programma si ferma in attesa di dati
- i caratteri battuti a tastiera vengono ricopiate lo stream cui, e vengono mostrati a video per eseguire eventuali correzioni
- tali caratteri faranno parte effettivamente di cui solo quando viene battuto il testo "ritorno carrello"

riduzione

al comando di esecuzione di un programma, lo stream cui può essere rediratto su un file "file.in" residente in memoria di massa:

- leggi.exe < leggi.in **valori letti i valori da un altro file**

lettura di caratteri

```
char c;  
cin >> c;
```

- se il carattere contenuto nella variabile mai è uno spazio bianco:

- viene prelevato il carattere e assegnato a c
- il puntatore si sposta verso la cella successiva

- se c'è uno spazio bianco:

- il carattere viene ignorato

- per leggere spazi bianchi:

- char c;
cin.get(c) **funzione member**

lettura interi / reali:

il lettore si sposta da una casella alla successiva finendo da dove caratteri vengono con il tipo della variabile, si ferma al primo carattere non preveduto dalle istruzioni

esempi:

- lettura: il lettore userà indirizzi una sequenza di caratteri vengono con la sintassi del tipo
- occorre rimediare con cin.clear()
- introdurremo una fine stream: Win (ctrl+z)

differenze con array

In più' pensare che cin>>c memorizza una parola, in realtà nel memoria viene prelevata una sola lettera, tuttavia seguendo un ciclo di una parola lettrale e una rapida uscita voglioso prelevare le lettere consecutive

lettura multiple

il corpo di un'instruzione lettura è costituito da un'espressione lettura che, oltre ad effettuare l'operazione indicata, restituisc a risultato, costituito dall'indirizzo dello stream coinvolto; per questo più' istruzioni lettura nello stesso stream possono essere sovrapposte. Esempio: cin>>a>>b;
obiettivo da riportare, cin>>a; cin>>b;

implementazione

quando devo lavorare con le classi ho bisogno di sovraccaricare l'operatore >>:

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

class Persona {
private:
    string nome;
    int eta;

public:
    // Funzione per stampare i dati
    void stampa() const {
        cout << "Nome: " << nome << ", Età: " << eta << endl;
    }

    // Sovraccarico dell'operatore >>    avendo il punto mantenendo la sequenza
    // non con cin::operator>> (cin)
    friend istream& operator>>(istream& in, Persona& p);
};

// Definizione dell'operatore >>    oggetto di classe istream, come nel esempio qui
istream& operator>>(istream& in, Persona& p) {
    // Legge il nome (una stringa) e L'età (un intero)
    in >> p.nome >> p.eta;
    return in; // Restituisce il flusso per il concatenamento
}

int main() {
    Persona p;
    cout << "Inserisci il nome e l'età della persona: ";
    cin >> p; // Usa l'operatore >> per leggere i dati in p

    p.stampa(); // Stampa i dati letti

    return 0;
}
```

ostream

```
class ostream
{ // stato: configurazione di bit
  // ....
public:
  ostream(){...};
  ostream& operator<<(int);
  ostream& operator<<(double);
  ostream& operator<<(char);
  ostream& operator<<(const char* );
  ostream& put(char c) scrittore in un stream uscita;
  ostream& write(const char* s, int n) richiede che n caratteri contenuti in s in uscita;
  ostream& operator<< (ostream& (*pf )(ostream&));
};
```

riassumi

basic-output-expression-statement

output-stream << expression;

esercizi

- calcolo dell'espressione (`cout << my`) e conversione in sequenze caratteri (`bool = 1, 0; enum = correttore`)
- trasformazione del primo carattere, allo stream

formato

- ampiezza del campo = è possibile visualizzare un dato visualizzando più o meno caratteri

```
int i = 42
cout << setw(10) << i; // 0000000042
```

- lunghezza parte frazionaria (per n. reali) = `setprecision(2)`

riduzione

come per cui, un file può essere ridotto su un file.out

- `scri: exe > file.out`

CONTROLLI SU STREAM

Lo stato di uno stream è fatto tenendo una serie di bit di stato che indicano la condizione del flusso di input e output, il campo dei dati è contenuto nella classe ios (contenuto in ostream) dichiarata in iostream, ostream

bit di stato

Lo stato è corretto (good) se tutti i bit valgono 0:

- bit fail: 1, ogniqualvolta si verifica un errore recuperabile
- bit bad: 0 recuperabile, 1 non (non permette operazioni successive)
- bit eof: (end of file) 1 quando viene raggiunta la marca di fine stringa

operatori

intuizione sui valori booleani:

- fail(): true se almeno uno tra fail e bad è 1
- eof(): bit eof = 1
- good(): bit tutti = 0

funzione clear

La funzione clear() serve per ripristinare lo stato di uno stream, può essere usata per reimpostare i bit di stato a una condizione specifica, generalmente chiamate quando vogliamo continuare ad usare lo stream dopo un errore rilevato

enumeratori

Le costanti in cui voleva un singolo bit di stato sono costituite da enumeratori dichiarati nella classe ios:
possiamo combinare i vari bit scegliendo utilizzando operatori bit-bit

un come condizionale

- è possibile inserire un dentro un if, perché vengono volutamente i bit di stato (se il bit non è 0)
- non è possibile inserire una negazione = $\text{if}(\text{!uu})$ perché non esiste tra i vari errori, sarebbe più corretto gettando con i bit di stato

MANIPOLAZIONE DEL FILE

le librerie di input/output consentono di utilizzare stream da varie fonti associate a file esterni dal sistema operativo, includendo <fstream.h>

sintassi

basic-file-stream-definition

fstream identifier-list; stream iogr, usc;

aperture chiuse

1). open(): • associa uno stream ad un file, apertura dello stream

• modalità di lettura rappresentate dalle costanti.

- lettura = ios::in

 ↳ ios:: corrente

ES. • iogr.open("file1.in", ios::in) ↳ costante

- scrittura = ios:: out

- modifica file file: ios::out | ios:: app ↳ append

- file non esiste viene creato, puntatore su 1 casella

• usc.open("file2.out", ios::out)

- file se non esiste viene creato, puntatore su 1 casella → viene visualizzata solo una linea
stream, eventuali dati vengono perduti

• usc.open("file.out", ios::out | ios:: app)

- file se non esiste viene creato, puntatore su ultime caselle → dati non perduti

• una volta aperto lo stream gli operatori non gli stessi dello stream standard,
le operazioni vengono effettuate sul file associato ES: usc << x; iogr >> x

2). close(): • chiude uno stream aperto una volta utilizzato

ES: • usc.close()

• iogr.close()

• una volta chiuso lo stream può essere riaperto in associazione a qualunque altro file

• a fine programma tutti gli stream vengono chiusi

errori

quanto detto per cui vale anche per qualsiasi file aperto in lettura

struttura dati manipolazione file

filestream, ifstream, ofstream hanno una struttura simile a quelle viste per gli stream input/output, utilizzando i bit di stato

cosa ci faccio con queste informazioni?

quindi ci dovranno verificare degli errori; oppure usare gli altri, con le funzioni membri associate ai bit di stato

RIDEFINIZIONE OPERATORI INGRESSO/ USCITA

permette di utilizzare oggetti della tua classe nei flussi input/output esattamente come fai con i tipi primitivi

overloading

vorrei una ridefinizione degli operatori >>, <<

- devono essere implementate globalmente, accedendo alle funzioni pubbliche della classe
- output: ostream & operator << (ostream & os, complesso & z) {
 os << 'l' << z.reale << ',' << z.imm << 'r';
 return os; }
funzioni da accedono alle poste private
- input:

```
istream& operator>>(istream& is, complesso& z) {  
    double re = 0, im = 0; // Variabili temporanee per leggere i valori  
    char c = 0;  
  
    is >> c; // Legge il primo carattere (deve essere '(')  
    if (c != '(') {  
        is.clear(ios::failbit); // Imposta il flag failbit se il formato è errato  
    } else {  
        is >> re >> c; // Legge la parte reale e il separatore ','  
        if (c != ',') {  
            is.clear(ios::failbit); // Imposta failbit se manca la virgola  
        } else {  
            is >> im >> c; // Legge la parte immaginaria e la parentesi ')'  
            if (c != ')') {  
                is.clear(ios::failbit); // Imposta failbit se manca la parentesi chiusa  
            }  
        }  
    }  
  
    z = complesso(re, im); // Aggiorna l'oggetto `z` solo se la lettura è valida  
    return is; // Restituisce il flusso  
}
```

possette di fare:

c1 >> c2

mette where funzione costruttori

OPERATORI DI CONVERSIONE

convertire un oggetto classe in un valore di un altro tipo

esempio:

```
class complesso
{
    double re, im;
public:
    operator double ()
```

.cpp

```
compleSSO::operator double () { return re + im; }
```

```
compleSSO :: operator int () { return static_cast <int> (re + im); }
```

```
int main () { compleSSO c1 (3,2);
    cout << (double) c1 << endl; // 7,2
    cout << (int) c1 << endl; // 7
```

non posso usare static cast senza un overload perché non c'è una overload

OVERLOADING & OPERATORE DI ASSEGNAZIONE

quando si hanno classi che operano in memoria dinamica e che quindi sono caratterizzate dalle varie di puntatori, l'operatore di assegnamento predefinito non è sufficiente

problema

effettuando l'assegnamento di due oggetti diverse, effettuerà una copia tra membro e membro dei valori degli oggetti diverse, e quindi dei puntatori.

Ottiene due puntatori che puntano alle stesse stringhe, e verrà effettuato un doppio **delete** all'uscita dello scope ① ②

soluzione

```
#include "stringa.h"
#include <cstring> // Per usare strcpy

// Costruttore
stringa::stringa(const char s[]) {
    str = new char[strlen(s) + 1]; // Alloca memoria dinamica
    strcpy(str, s); // Copia la stringa
}

// Distruttore
stringa::~stringa() {
    delete[] str; // Dealloca la memoria
}

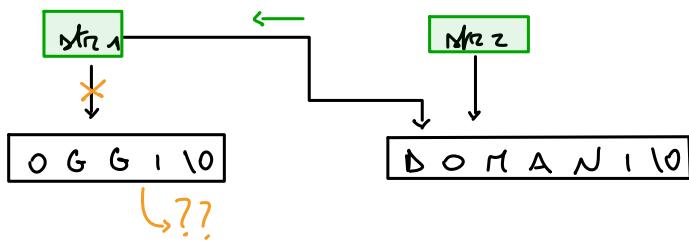
// Costruttore di copia
stringa::stringa(const stringa& other) {
    str = new char[strlen(other.str) + 1]; // Alloca memoria per la copia
    strcpy(str, other.str); // Copia il contenuto
}

// Operatore di assegnamento
stringa& stringa::operator=(const stringa& other) {
    if (this != &other) { // Verifica se non è un auto-assegnamento
        delete[] str; // Libera la memoria già allocata
        str = new char[strlen(other.str) + 1]; // Alloca nuova memoria
        strcpy(str, other.str); // Copia il contenuto
    }
    return *this; // Restituisce il riferimento all'oggetto corrente
}
```

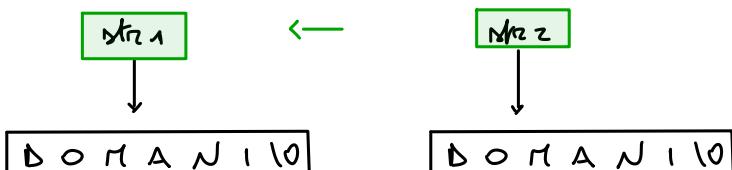


Mohi

- prine :



- dopo :



operatori deve

- 1) deallocare memoria dinamica dell'operando nostro
 - 2) allocare la memoria della dimensione dell'operando storico
 - 3) copiare i numeri storo e gli elementi della lista

contratualizar copiar o experimentar?

Caratteristica	Costruttore di copia	Operatore di assegnamento
Quando viene invocato?	Durante la creazione di un oggetto copia.	Durante l'assegnazione di un oggetto a un altro.
Cosa fa?	Crea un nuovo oggetto come copia dell'originale.	Assegna i valori di un oggetto a un altro già esistente.
Cosa gestisce?	Inizializza la memoria per un nuovo oggetto.	Gestisce l'auto-assegnamento, dealloca la memoria precedente e assegna nuovi valori.
Gestione della memoria dinamica	Deve allocare memoria per una nuova copia dei dati.	Deve liberare la memoria preesistente e allocare nuova memoria per i dati.

quelli che getti non uguali

due effetti non lo stesso effetto quando hanno lo stesso indirizzo

CONSTANTI E RIFERNIMENTI IN CLASSI

luce di inizializzazione
classe zelloriana d'uso

- quando un campo dati avesse definito costante, mantenere lo stesso valore per tutti gli oggetti classe, static
- un campo dati può avere const. e deve essere inizializzato al momento in cui viene definito un oggetto

problema

class complesso:

```
const int = re;  
int = im;
```

public:

```
compleks (int z, int i) {
```

```
im = i; re = z }
```

non ha più senso perdere istanza specifica a re, un valore

inizializzazione

compleks (int z, int i) : re(z) {
im = i } , viene inizializzato minore dell'oggetto
venerdì seguito dopo re(z) luce di inizializzazione

int main() {

```
compleks c1(5) x. c1 = 6
```

riferimento.

• voglio inizializzare prendendo i valori che un'altra inizializza:

, se non ti fasse, re potrebbe essere modificato

compleks:: compleks (const compleks & c) : re(c.re) { im = c.im } in questo caso dovrò passare alle due metà un compleks da venire inizializzato con tali valori

MEMBRI CLASSE

in una classe (classe principale) possono essere presenti: membri di tipo classe (classe secondaria) ≠ della principale

class struttura	class record
{	{ stringe nome; intente 1
. - - -	stringe cognome; intente 2
	public:

cosa accade?

- 1) quando un oggetto record viene creato, i costruttori delle classi secondarie vengono richiamati prima del corpo del costruttore della classe principale nell'ordine di dichiarazione:

record z; 1) costruttore nome
2) costruttore cognome
3) costruttore record

- 2) quando un oggetto della classe principale viene distrutto, i distruttori vengono chiamati nell'ordine inverso rispetto a quello in cui i membri classe sono dichiarati:

- 1) viene chiamato il distruttore della classe principale
- 2) vengono chiamati i distruttori nell'ordine inverso alla dichiarazione

costruttore

```
// file record.h
#include "stringa.h"

class record {
private:
    stringa nome; // Membro della classe record
    stringa cognome; // Membro della classe record
public:
    record(const char n[], const char c[]); // Costruttore della classe record
};

// file record.cpp
record::record(const char n[], const char c[])
: nome(n), cognome(c) // Inizializza i membri nome e cognome con le stringhe passate
{ // vengono inizializzati nome e cognome (campo dati)
    // Corpo del costruttore
}
```

int main() {

record pers ("mario", "rossi") // inizializzazione, creazione di due stringhe nome
stringe cognome

costruttore default

- se alcune classi ricevono solo un costruttore default, questi vengono chiamati.
- se tutte le classi hanno un costruttore default, anche quello della classe principale può essere un costruttore default

ARRAY DI OGGETTI CLASSE

un array può avere elementi come oggetti classe, se nella classe vengono definiti costruttori e distruttori, questi vengono invocati per ogni elemento dell'array:

```
campanello ( int i, int z ) {
    in = i ; re = z }

int main() {
    campanello vc [3]; // vc è un array dell'array
    for ( i=0 , i<3; i++ )
        vc [i].suona();
```

Initializzazione

- esplicate:

completo $vc[3] = \{ complesso(3,4), complesso(5,6), complesso(7,8) \}$

- implicite: si dichiara una funzione leggera che include: cin >> re; cin >> im, con un for visto e riempire l'array

NB: se la lista dei costruttori (cioè i costruttori di ogni oggetto clonato nell'array) non è completa, nella classe deve essere definito un costruttore default, che viene richiamato per gli oggetti apertamente mai inizializzati

MEMBRI STATICI

a volte succede che una classe ne ricopra delle informazioni globali, cioè appartenenti alla classe nel suo complesso e non alla singola istanza

es. volta istante, ogni chiamata volerà incremento di 1

esistenza

un campo dato statico esiste in un'unica copia di memoria

- nella classe viene solo dichiarato
- la definizione avviene al di fuori in un file .cpp e la memoria viene allocata

NB: - se definito nella parte pubblica posso accedere con . per un singolo oggetto (rimuovere un controllore generale)
- se definito nella parte privata sarà accessibile tramite funzione

esempio

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

class entity {
    int dato; // Un membro di istanza, diverso per ogni oggetto
public:
    static int conto; // Membro statico, condiviso da tutte le istanze della classe
    entity(int n); // Costruttore che inizializza 'dato' e incrementa 'conto'
};

// Definizione della variabile statica 'conto' globale
int entity::conto = 0;

entity::entity(int n) { // Definizione costruttore con oggetto contatore
    conto++; // Incrementa il contatore ogni volta che un oggetto viene creato
    dato = n; // Inizializza il membro 'dato' con il valore passato al costruttore
}

int main() {
    entity e1(1), e2(2), e3(3); // Creazione di tre oggetti
    cout << "Numero istanze: " << entity::conto << endl; // Stampa il numero di istanze
    return 0;
}
```

FUNZIONI CONST

funzione

- garantisce che non modifichi lo stato dell'oggetto su cui viene dichiarata
- può essere dichiarata su oggetti dichiarati come const, mentre le funzioni non const non possono

metodo

```
int getdato() {  
    return dato; } // stato variele const int dato nel campo dati:
```

può accadere di const ma non modificarelo

meccanismo overloading

se in una classe ci sono due stesse funzioni che dichiarano lo stesso nome ed una delle due è const, accade:

- in un oggetto const viene dichiarata la versione funzione const
- in un oggetto non const possono essere dichiarate entrambe ma viene data la priorità a quelle non const

NB: se dichiara e redene e il compilatore mette che il valore è const, invocherà automaticamente la funzione di dichiarazione const

CONVERSIONE CON COSTRUTTORI

rispondendo che non sia permessa la scrittura tra un double e un oggetto classe, ciò è ovviamente possibile se viene definito il costruttore di classe come precedentemente illustrato:

complesso operator + (complesso & x) {
 im = x.im ; re = x.re }

in questo modo quando scrivo un double a un'istanza double, viene riconosciuto
il numero scritto come un'altra istanza

PREPROCESSORE

parte del compilatore che esegue operazioni sul codice prima che inizi l'esecuzione binaria elettiva del programma:

- 1) può includere altri file: con `#include` puoi aggiungere il contenuto di file esterni
- 2) espansione dei simboli: usando la direttiva `#define` per creare macro o costanti
- 3) inclusione o esclusione condizionale (vedi dopo)

Inclusione file header

- 1) percorso a partire da cartella standart `#include <file.h>`
- 2) percorso a partire dalla cartella in un corrente la compilazione o percorso assoluto

- ES:
- `#include "file.h"` file nella stessa cartella progetto
 - `#include "subdir\file.h"` file in una cartella nella stessa cartella progetto
 - `#include "c:\subdir\file.h"` file con percorso assoluto

Definizione di simboli con precedenza un valore

```
#define ABDUL: 42
```

```
int main() {  
    int x = ABDUL // int x = 42
```

! = non consiglia di usare questa tecnica, è meglio scrivere `int x=42`, poiché il compilatore può effettuare tutti i controlli, se non ha un'informazione e in occasioni problematiche

macro

Una macro è un simbolo che prima della compilazione viene sostituito da una sequenza di elementi (caratteri corrispondenti alla sua definizione).

ES: `#define MAX(A,B) ((A)>(B)?(A):(B))` attenzione alle parentesi !!

Compilazione condizionale

RIDEFINIZIONE OPERATORE UNARIO

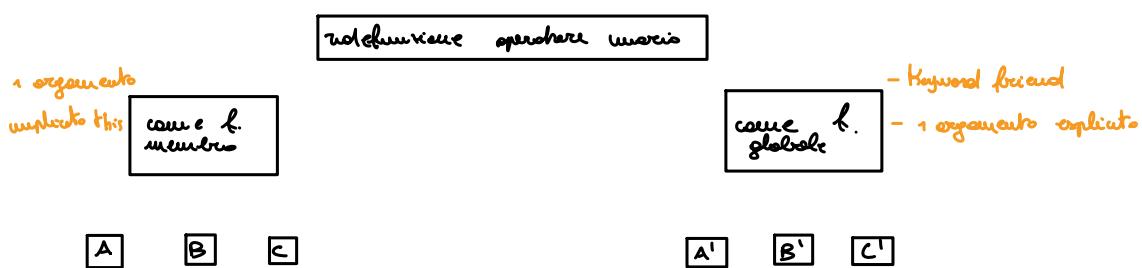
obiettivo

estendere ai tipi diversi operatori definiti tra tipi fondamentali:

strutture

una funzione con operatore (numero) (

modi di riferimento



procedimento

1) scegliere tra f. membro o globale (per operatore unario e= uguale)

2) scelta fra le soluzioni di una famiglia:

- `A, A'` = modificherà l'oggetto su cui operano e restituiranno void
- `B, B'` = non modificherà l'oggetto, restituiranno un nuovo oggetto come risultato dell'operazione
- `c, c'` = modificherà l'oggetto, restituirà un riferimento all'oggetto modificato

OSS. operatori: `--`, `++` → operator `++` il compilatore non copierà né è pre o post

- se è pre: operator `++(...)`
- se è post: operator `++(int)` il compilatore con lo comprende