

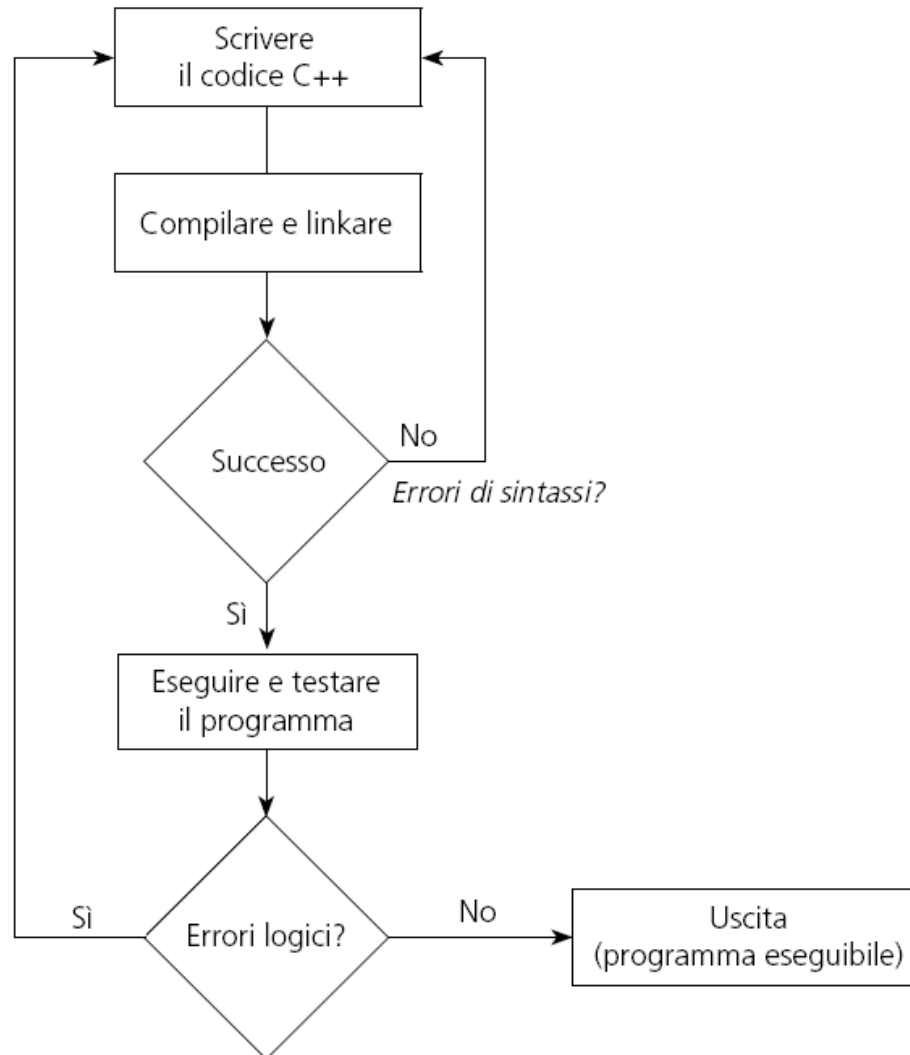
Il linguaggio C++

- **Bjarne Stroustrup** è l'inventore del C++ (1983), standardizzato nel 1998 (C++98), attualmente C++20
- Il C++ è un linguaggio di programmazione che supporta:
 - **programmazione procedurale:** rappresenta l'approccio tradizionale alla programmazione (linguaggio C). E' basata su una metodologia di decomposizione funzionale, ovvero consiste nel suddividere un algoritmo in sottoprogrammi (funzioni) che vengono richiamati all'occorrenza.
 - **programmazione orientata agli oggetti** (OOP, Object Oriented Programming): organizza il codice sotto forma di classi, ossia il codice è costituito da oggetti software (variabili istanze di classi) che interagiscono tra di loro.
 - **programmazione generica:** è un paradigma di programmazione più astratto che consente di definire funzioni e classi che possano essere utilizzati con tipi di dati diversi

Alcune regole generali

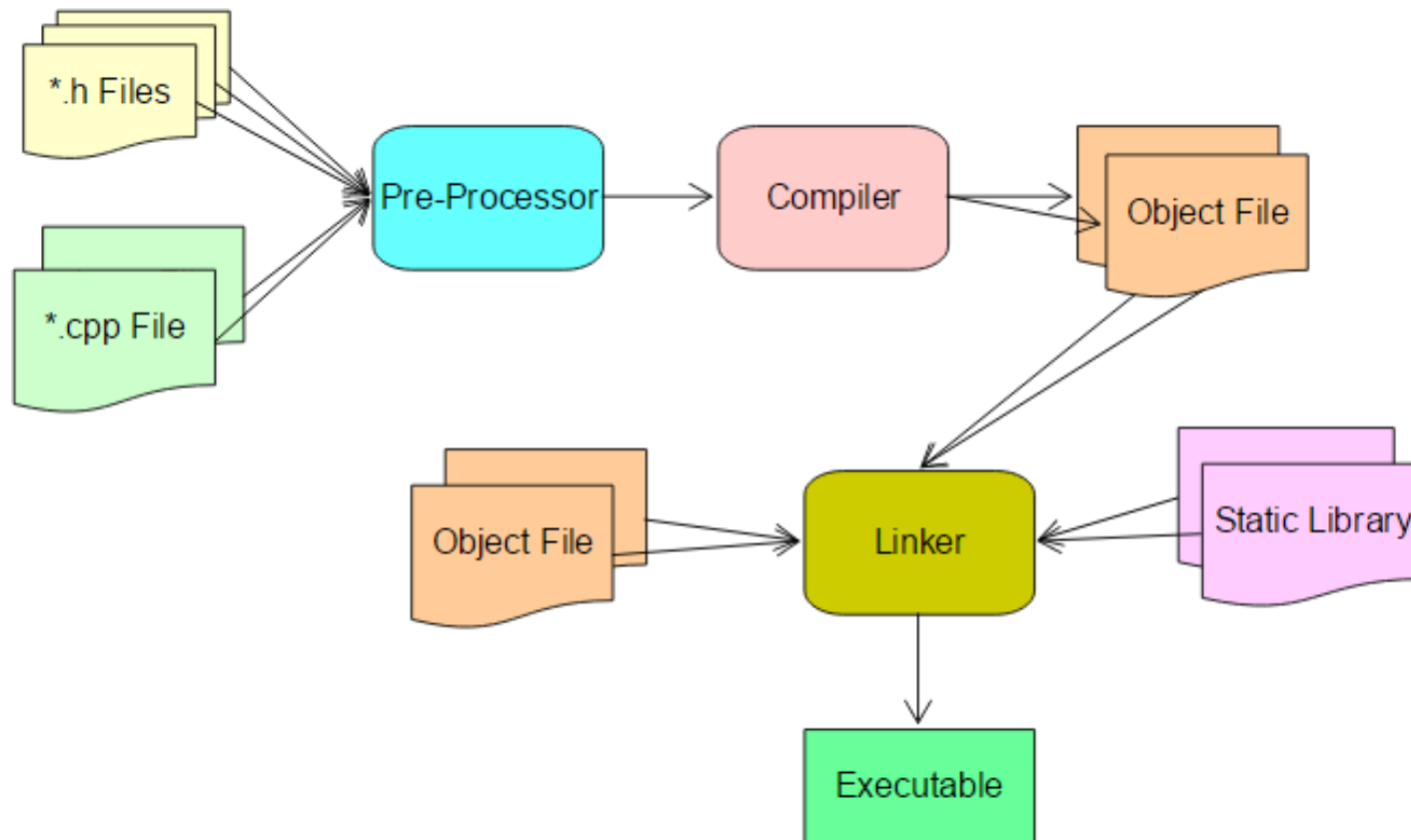
- La **maggior parte** delle istruzioni deve terminare con un punto e virgola (;)
- Commenti
 - `/*`
 <testo>
`*/`
 - `//` testo (fino a fine riga)
- Il C++ è un linguaggio **case-sensitive** (c'è differenza tra caratteri minuscoli e maiuscoli)

Costruzione di un programma



Flusso di compilazione

- C++ è un linguaggio compilato
- I nostri progetti saranno costituiti da più files sorgenti (.cpp) e files di header (.h, .hpp)
- L'utilizzo di un ambiente di sviluppo integrato nasconde il flusso di compilazione



Pre-processor

- gestisce le **direttive di preprocessor** (righe che iniziano con un carattere #)
- le **direttive di preprocessor** non devono terminare con un punto e virgola
- gestisce l'inclusione di file header (direttiva **#include**)

`#include <nomefile>`

sostituisce la direttiva `#include <nomefile>` con una copia del file della libreria standard

`#include "nomefile"`

sostituisce la direttiva `#include "nomefile"` con un file header definito dall'utente in un percorso assoluto o relativo

- gestisce la definizione di costanti simboliche e macro (direttiva **#define**)

`#define ELEMENTS 100`

Sostituisce in tutto il programma 100 alla parola ELEMENTS

Pre-processor

- gestisce la compilazione condizionale del codice
(**#ifdef, #ifndef, #if, #endif, #else**)
Applicazioni: debug, guardia contro inclusione multipla

Esempio di uso per debug:

```
#define DEBUG
```

```
.....
```

```
.....
```

```
#ifdef DEBUG
```

```
    cout << "This is the test version, i=" << i << endl; //questa parte di codice viene eseguita
```

```
#else
```

```
    cout << "This is the production version!" << endl; // questa parte non viene eseguita
```

```
#endif
```

Compilatore

- Converte i file sorgenti in linguaggio macchina
- Genera i file oggetto (.o, .obj)

Linker

- Unisce i file oggetto e risolve indirizzi/riferimenti
- Associa eventuali librerie esterne pre-compilate
- Risultato: file eseguibile .exe (in Windows)

Scheletro programma C++

Esempio di un programma costituito da un solo file sorgente che contiene il main del programma. La funzione main() viene eseguita automaticamente quando si esegue il programma.

```
// my first program in C++
#include <iostream>
int main() // funzione main punto di ingress del programma
{
    std::cout << "HelloWorld!";
    return 0;
}
```


Errori di compilazione e warning

- **Warning:**

- messaggi di avvertimento
- non interrompono la compilazione
- avvisano della (possibile) presenza di irregolarità nel codice

cppfile.cpp:5: warning: unused variable 'float fValue'

- **Errori:**

- interrompono la compilazione
- indicano errori che devono essere corretti
- **importante** saper **interpretare i messaggi di errore del compilatore**

day.cpp:25: 'DayOfYear' undeclared (first use this function)

Valore e Tipo dell'informazione

- In C++ (così come negli altri linguaggi di programmazione) qualsiasi informazione è definita tramite due caratteristiche fondamentali:
 - **Valore**: indica il contenuto associato all'informazione
 - **Tipo**: indica l'insieme dei valori che un dato può assumere
- Esempi:

Valore	Tipo
3.5	reale
5	intero
"Luigi Rossi"	stringa di caratteri

- In un programma l'informazione può essere organizzata in:
variabili, costanti, espressioni

Tipi di dato

- Il tipo di un dato oltre a specificare l'insieme dei valori che costituiscono il tipo definisce anche l'insieme delle operazioni ammissibili sugli elementi del tipo
- Tutte le costanti, variabili ed espressioni appartengono ad un certo tipo
- Per definire il tipo di dato associato ad una variabile o ad una costante (prima del loro utilizzo) è necessario effettuare una **dichiarazione** della variabile o della costante.
- Un tipo di dato si dice ordinato se è definita una relazione d'ordine tra i suoi elementi.
- Per ogni variabile o costante che viene dichiarata il suo tipo di dato consente al compilatore di determinare a priori la quantità di memoria necessaria per memorizzare la variabile o la costante.

Tipi di dato elementari

- **tipo di dato INTERO (int)**

È costituito da un sotto insieme limitato dei numeri interi.

- **tipo di dato REALE (float o double)**

È costituito da un sotto insieme limitato e discreto dei numeri reali

Operazioni ammesse

assegnazione =

somma +

sottrazione -

moltiplicazione *

divisione intera /

confronto >, <, >=, <=, ==, !=

Test di uguaglianza e assegnazione

- In C++ l'espressione booleana $a==b$ indica il valore di verità dell'affermazione "a e b sono uguali"
- $a==b$ è true se le due variabili a e b (o espressioni) hanno lo stesso valore, e false se hanno valori diversi
- Es: $2+2==4$ vale true $2+2==5$ vale false
- $a=b$ invece indica una operazione di assegnamento: "assegna alla variabile a il valore della variabile b"

Un errore molto comune è confondere $(a==b)$ con $(a=b)$, soprattutto all'interno di una espressione `if(...)`, per esempio scrivere `if(x=3)` al posto di `if(x==3)`

Tipi di dato elementari

- **tipo di dato CARATTERE (char)**

È costituito da un insieme di caratteri, alcuni stampabili (caratteri alfabetici, cifre, caratteri di punteggiatura, ecc.) ed altri non stampabili.

I sotto-insiemi delle lettere e delle cifre sono ordinati.

Per la rappresentazione in memoria, viene tipicamente usato il codice ASCII, che rappresenta ogni carattere con un numero intero.

Il tipo carattere è di fatto un dato di tipo numerico sul quale sono permesse tutte le operazioni definite sul tipo intero.

Tipi di dato elementari

Tabella caratteri in codice ASCII

Dec	Bin	Hex	Char	Dec	Bin	Hex	Char	Dec	Bin	Hex	Char	Dec	Bin	Hex	Char
0	0000 0000	00	[NUL]	32	0010 0000	20	space	64	0100 0000	40	@	96	0110 0000	60	`
1	0000 0001	01	[SOH]	33	0010 0001	21	!	65	0100 0001	41	A	97	0110 0001	61	a
2	0000 0010	02	[STX]	34	0010 0010	22	"	66	0100 0010	42	B	98	0110 0010	62	b
3	0000 0011	03	[ETX]	35	0010 0011	23	#	67	0100 0011	43	C	99	0110 0011	63	c
4	0000 0100	04	[EOT]	36	0010 0100	24	\$	68	0100 0100	44	D	100	0110 0100	64	d
5	0000 0101	05	[ENQ]	37	0010 0101	25	%	69	0100 0101	45	E	101	0110 0101	65	e
6	0000 0110	06	[ACK]	38	0010 0110	26	&	70	0100 0110	46	F	102	0110 0110	66	f
7	0000 0111	07	[BEL]	39	0010 0111	27	'	71	0100 0111	47	G	103	0110 0111	67	g
8	0000 1000	08	[BS]	40	0010 1000	28	(72	0100 1000	48	H	104	0110 1000	68	h
9	0000 1001	09	[TAB]	41	0010 1001	29)	73	0100 1001	49	I	105	0110 1001	69	i
10	0000 1010	0A	[LF]	42	0010 1010	2A	*	74	0100 1010	4A	J	106	0110 1010	6A	j
11	0000 1011	0B	[VT]	43	0010 1011	2B	+	75	0100 1011	4B	K	107	0110 1011	6B	k
12	0000 1100	0C	[FF]	44	0010 1100	2C	,	76	0100 1100	4C	L	108	0110 1100	6C	l
13	0000 1101	0D	[CR]	45	0010 1101	2D	-	77	0100 1101	4D	M	109	0110 1101	6D	m
14	0000 1110	0E	[SO]	46	0010 1110	2E	.	78	0100 1110	4E	N	110	0110 1110	6E	n
15	0000 1111	0F	[SI]	47	0010 1111	2F	/	79	0100 1111	4F	O	111	0110 1111	6F	o
16	0001 0000	10	[DLE]	48	0011 0000	30	0	80	0101 0000	50	P	112	0111 0000	70	p
17	0001 0001	11	[DC1]	49	0011 0001	31	1	81	0101 0001	51	Q	113	0111 0001	71	q
18	0001 0010	12	[DC2]	50	0011 0010	32	2	82	0101 0010	52	R	114	0111 0010	72	r
19	0001 0011	13	[DC3]	51	0011 0011	33	3	83	0101 0011	53	S	115	0111 0011	73	s
20	0001 0100	14	[DC4]	52	0011 0100	34	4	84	0101 0100	54	T	116	0111 0100	74	t
21	0001 0101	15	[NAK]	53	0011 0101	35	5	85	0101 0101	55	U	117	0111 0101	75	u
22	0001 0110	16	[SYN]	54	0011 0110	36	6	86	0101 0110	56	V	118	0111 0110	76	v
23	0001 0111	17	[ETB]	55	0011 0111	37	7	87	0101 0111	57	W	119	0111 0111	77	w
24	0001 1000	18	[CAN]	56	0011 1000	38	8	88	0101 1000	58	X	120	0111 1000	78	x
25	0001 1001	19	[EM]	57	0011 1001	39	9	89	0101 1001	59	Y	121	0111 1001	79	y
26	0001 1010	1A	[SUB]	58	0011 1010	3A	:	90	0101 1010	5A	Z	122	0111 1010	7A	z
27	0001 1011	1B	[ESC]	59	0011 1011	3B	;	91	0101 1011	5B	[123	0111 1011	7B	{
28	0001 1100	1C	[FS]	60	0011 1100	3C	<	92	0101 1100	5C	\	124	0111 1100	7C	
29	0001 1101	1D	[GS]	61	0011 1101	3D	=	93	0101 1101	5D]	125	0111 1101	7D	}
30	0001 1110	1E	[RS]	62	0011 1110	3E	>	94	0101 1110	5E	^	126	0111 1110	7E	~
31	0001 1111	1F	[US]	63	0011 1111	3F	?	95	0101 1111	5F	_	127	0111 1111	7F	[DEL]

Tipi di dato elementari

#include <limits>

per utilizzare le costanti predefinite

Tipo	Dimensione in bytes	Valore Minimo...Valore Massimo
char	1	CHAR_MIN=-128...CHAR_MAX=127
short	2	SHRT_MIN=-32768...SHRT_MAX=32767
int	4	INT_MIN=-2147483648...INT_MAX=2147483647
unsigned int	4	0...UINT_MAX=4294967295
float	4	$1.7 * (10^{-38}) \dots 3.4 * (10^{38})$
double	8	$2.2 * (10^{-308}) \dots 1.7 * (10^{308})$

Tipi di dato elementari

- **tipo di dato LOGICO (bool)**

È un tipo costituito da due valori (indicati con **false** e **true**), viene utilizzato per gestire le informazioni di tipo logico (es. il risultato di un confronto).

- In C++ il valore **0** equivale a FALSE e qualunque valore **diverso da zero** equivale a TRUE.

Operazioni ammesse

assegnazione =

OR logico ||

AND logico &&

negazione !

Tipi di dato elementari

AND

OR

x	y	x && y	x y
false	false	false	false
false	true	false	true
true	false	false	true
true	true	true	true

NOT

x	!x
false	true
true	false

Parole chiave (riservate)

asm	double	mutable	struct
auto	else	namespace	switch
bool	enum	new	template
break	explicit	operator	this
case	extern	private	throw
catch	float	protected	try
char	for	public	typedef
class	friend	register	union
const	goto	return	unsigned
continue	if	short	virtual
default	inline	signed	void
delete	int	sizeof	volatile
do	long	static	wchar_t
while			

Variabili

- una variabile è una entità appartenente ad un certo tipo, ed è identificata da un nome (identificatore). Il valore di una variabile può essere sia utilizzato (lettura) che modificato (scrittura).
 - definizione: ***<tipo> nome_variabile;***
 - `int numero_di_giorni;`
 - `char lettera;`
 - `float tasso_variabile;`
 - punto di definizione
 - inizio di un file

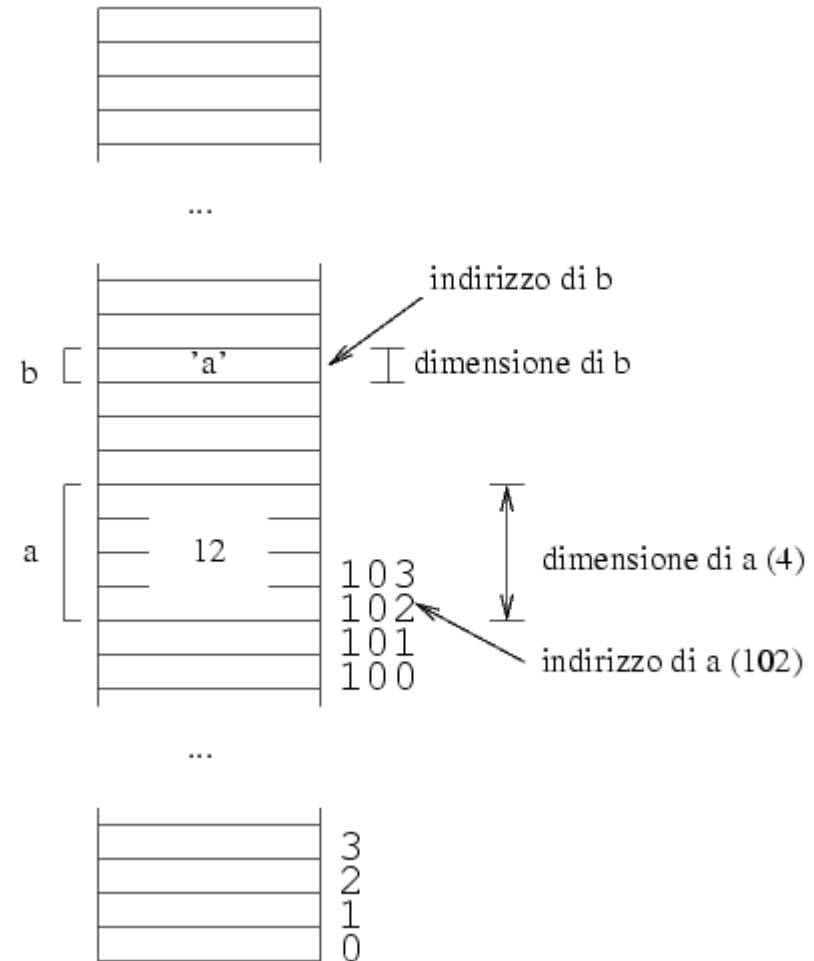
```
#include <iostream>
int i;
```
 - inizio di un blocco di codice (es: `main()`, funzione, metodo, classe)

```
{ int i; ... }
```
 - punto in cui si utilizza la variabile

```
for(int i=0; i < 10; i++)
```

Variabili

- Più in particolare una variabile è caratterizzata da un nome (identificatore), da un tipo, e da un indirizzo di memoria.
- L'indirizzo di memoria si riferisce alla prima cella della memoria del computer che contiene la variabile (una variabile può occupare più di una cella di memoria, le celle di memoria hanno dimensione 1 byte)
- Il tipo di dato della variabile determina la dimensione (numero di bytes) occupati dalla variabile



Variabili

- è possibile al momento della definizione di una variabile attribuire alla variabile un valore di inizializzazione, esempio:

```
int i=10;
```

- una variabile in C++ ha sempre un valore, se la variabile non viene inizializzata il suo valore sarà un valore di default
- Il valore di una variabile in un programma può essere modificato tramite una istruzione di assegnazione che utilizza l'operatore di assegnamento

variabile = espressione

- <variabile> si dice essere un "left value" (cioè l'indirizzo di una variabile), <espressione> è un "right value" (cioè un valore)

Esempi:

```
y=10;
```

Assegnamenti multipli: **Y = X = 3;**

Standard input/output

In C++ le operazioni di I/O sono fatte operando su “stream” di bytes. Uno stream è una sequenza di byte.

- cin: standard input (tastiera)
- cout: standard output (finestra del terminale sullo schermo)

```
#include<iostream>
int main() {
    int x,y;
    std::cout<< "Enter two integers numbers:";
    std::cin>> x >> y;
    std::cout<< "La somma di "<< x << " e "<< y << " vale: "<<
(x+y) <<std::endl;
    system("pause");
    return 0;
}
```

Standard input/output

- L'output si effettua applicando l'operatore di inserimento << a cout, cout è un oggetto della classe ostream
- L'operatore << inserisce il valore alla sua destra nel flusso di dati alla sua sinistra, più operatori << si possono concatenare

```
std::cout << "Salve" << " a " << "tutti"<< std::endl;
```

- endl emette un carattere di ritorno a nuova linea
- L'input si effettua applicando l'operatore di estrazione >> a cin, cin è un oggetto della classe istream, la chiamata a cin è bloccante, l'input viene elaborato dopo aver premuto il tasto invio
- L'operatore >> deve essere seguito da una variabile, più operatori >> possono essere concatenate, i valori inseriti devono essere separati da uno o più spazi/tabulazioni/newline

```
std::cin >> a >> b;
```


Durata e visibilità di una variabile

- La visibilità (scope) di una variabile si estende dal punto in cui è definita fino al limite del blocco d'istruzioni {...} in cui è definita. Le variabili possono essere locali o globali.
- **variabili locali ad un blocco di codice:**
 - non possono essere modificate da istruzioni esterne al blocco in cui sono definite
 - blocchi diversi possono utilizzare nomi identici per le proprie (distinte) variabili locali
 - una variabile locale non esiste in memoria fino a che non viene eseguito il blocco di istruzioni in cui è definita, e termina di esistere quando viene eseguita l'ultima istruzione del blocco

```
{  
    float temperatura = 75.45;  
    int i;  
    int j = 5;  
    int risposta;  
    i = j; // assegna alla variabile i il valore della variabile j  
    risposta = (i + 4) * 5; // alla variabile viene assegnato il valore di una  
                          // espressione  
}
```

Durata e visibilità di una variabile

- **variabili locali ad un blocco di codice:**

All'interno di ogni blocco di codice {...} sono visibili:

- Le variabili definite al suo interno
- Le variabili definite nel blocco che eventualmente lo contiene

La definizione di una variabile in un blocco di codice annulla la visibilità di eventuali altre variabili con lo stesso nome, ma definite in blocchi esterni.

Le variabili definite in un blocco di codice si definiscono variabili **automatiche** poiché vengono allocate all'inizio dell'esecuzione del blocco di codice e vengono deallocate alla fine del blocco di codice. Le variabili automatiche (a differenza delle variabili statiche che vedremo) non conservano il loro valore tra due esecuzioni successive dello stesso blocco di codice.

Durata e visibilità di una variabile

- **variabili locali ad un blocco di codice:**

```
#include <iostream>
int main()
{
    int x= 10;
    int z = 35;
    std::cout << x << std::endl;
    {
        int x= 20;
        int y = 5;
        std::cout << x << " " << y << " " << z << std::endl;
    }
    std::cout << x << std::endl;
    //std::cout << y << std::endl; //errore
    return 0;
}
```

Durata e visibilità di una variabile

- **variabili globali:**

- variabili che si dichiarano prima (al di fuori) del main() e sono visibili ovunque nel programma, tranne che nei blocchi in cui esistono variabili locali con lo stesso nome

```
#include <iostream>
int a, b, c; //definizione di variabili globali
main()
{
    int d; // questa una variabile locale
}
```

- L'uso di variabili globali è **fortemente sconsigliato**

Esempi di uso di variabili

```
int i, j, val_m;
```

```
const int ci = i;
```

```
2040 = val_m; // errore; 2040 non ammissibile come left value
```

```
i + j = val_m; // errore; i + j non ammissibile come left value
```

```
ci = val_m; // errore; ci non ammissibile come left value
```

```
i = j; // corretto
```

Esempi di uso di variabili

- **Esempio di uso di variabili bool:**

```
int ore = 4; int minuti = 21; int secondi = 0;  
bool timeIstue = ore && minuti && secondi;
```

poiché il risultato deriva dall'esame dei tre operandi e c'è un valore (secondi) che è uguale a zero (ovvero equivale a false) il risultato dell'espressione è false.

- **tipo di dato void**

È considerato un tipo “vuoto”, non è possibile definire delle variabili di tipo void. Il tipo void viene utilizzato per descrivere delle funzioni che non hanno un valore di ritorno.

Deduzione automatica del tipo

- a partire dal C++11 è possibile dichiarare una variabile indicando genericamente **auto** come tipo, il tipo vero e proprio della variabile viene dedotto automaticamente dall'espressione che la inizializza.

```
#include <iostream>
int main() {
    auto y = 5;    // compilatore deduce che y è variabile int
    auto k = 2.2;
    auto s = "hello";
    std::cout << y << " " << k << " " << s << std::endl;

    return 0;
}
```

Costanti

- una costante è una entità appartenente ad un certo tipo, il cui valore rimane inalterato durante l'esecuzione del programma. Ad una costante può essere attribuito un nome.
- In C++, il qualificatore `const` permette di definire costanti

Esempi:

```
const int MESI = 12;  
const float PIGRECO = 3.141592;  
const char CARATTERE = 'a';
```


Espressioni

- **Espressioni:** un'espressione è una sequenza di operandi, operatori e parentesi, gli operandi possono essere variabili o costanti. Il tipo dell'espressione complessiva dipende dai tipi degli operandi coinvolti.

Esempi: (a,b variabili intere; x,y variabili reali)

$a*b+50$ è un'espressione di tipo intero

$a*3.1415$ è un'espressione di tipo reale

$x/2$ è un'espressione di tipo reale

$b/2$ è un'espressione di tipo intero (se $b=1$, il valore di $b/2$ è 0)

Assegnamento composto

op	uso	equivale a	descrizione
+=	<code>a += b</code>	<code>a = a + b;</code>	somma <i>a</i> e <i>b</i> ed assegna il risultato alla variabile <i>a</i>
-=	<code>a -= b</code>	<code>a = a - b;</code>	sottrae <i>b</i> ad <i>a</i> ed assegna il risultato alla variabile <i>a</i>
*=	<code>a *= b</code>	<code>a = a * b;</code>	moltiplica <i>a</i> per <i>b</i> ed assegna il risultato alla variabile <i>a</i>
/=	<code>a /= b</code>	<code>a = a / b;</code>	divide <i>a</i> per <i>b</i> ed assegna il risultato alla variabile <i>a</i>
%=	<code>a %= b</code>	<code>a = a % b;</code>	mette in <i>a</i> il resto della divisione intera di <i>a</i> per <i>b</i>

Operatori “bitwise” su variabili int

Op	Operazione	Esempio
&	AND <i>bit a bit</i>	01001001 ('I') 01001110 ('N') 01001000 ('I' & 'N' == 'H')
	OR <i>inclusivo bit a bit</i>	0000000010011000 (152) 0000000000000101 (5) 0000000010011101 (152 5 == 157)
^	OR <i>esclusivo bit a bit</i>	0000000001010011 (83) 0000000011001100 (204) 0000000010011101 (83 ^ 204 == 157)
<<	<i>Spostamento di tutti i bit verso sinistra</i>	0000000000000111 (15) 00000000000111100 (15 << 2 == 60)
>>	<i>Spostamento di tutti i bit verso destra</i>	00000000000111100 (60) 0000000000011110 (60 >> 1 == 30)

Operatori: precedenza e associatività

Nelle espressioni che contengono una successione di operazioni il risultato di ogni operazione diviene operando per le operazioni successive, fino a giungere ad un unico risultato. L'ordine in cui le operazioni sono eseguite è regolato da criteri di precedenza e associatività fra gli operatori.

Es: **a op1 b op2 c** (a, b, c sono operandi, op1 e op2 sono operatori)

1. se op1 ha precedenza il risultato di **a op1 b** diventa left-operand di **op2 c**
2. se op2 ha la precedenza: il risultato di **b op2 c** diventa right-operand di **a op1**
3. se **op1** e **op2** hanno la stessa precedenza, ma l'associatività procede da sinistra a destra, ci si riconduce al caso 1.
4. se **op1** e **op2** hanno la stessa precedenza, ma l'associatività procede da destra a sinistra: ci si riconduce al caso 2.

Per ottenere che un'operazione venga comunque eseguita con precedenza rispetto alle altre, bisogna racchiudere operatore e operandi fra parentesi tonde.

Es. **a op1 (b op2 c)** (la seconda operazione viene eseguita per prima)

Operatori: precedenza e associatività

Operatori a precedenza decrescente	Associatività
::	sinistra
() [] . -> ++ (postfisso) -- (postfisso) dynamic_cast static_cast reinterpret_cast const_cast typeid	sinistra
++ (prefisso) -- (prefisso) ~ ! - (unario) + (unario) & (indirizzione) * (indirizzo) sizeof new delete	destra
(tipo) (casting convenzionale)	destra
. * -> *	sinistra
* (moltiplicazione) / %	sinistra
+ (binario) - (binario)	sinistra
<< >>	sinistra
< <= > >=	sinistra
== !=	sinistra
& (bitwise and)	sinistra
^	sinistra
	sinistra
&&	sinistra
	sinistra
? : (espressione condizionale)	destra
= *= /= %= += -= <<= >>= &= = ^=	destra
, (operatore virgola)	sinistra

Istruzioni condizionali e iterative

- CONDIZIONALI

- if <expr.> { } else { }
- switch <expr.> { case <cost> : .. }
- operatore ternario

- ITERATIVE

- while <expr.> { }
- for (..) { }
- do { } while <expr.>

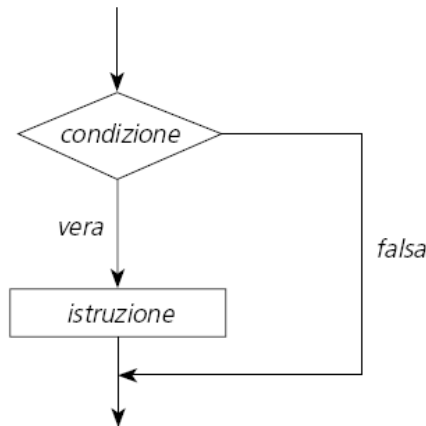
Istruzioni condizionali e iterative

- COSTRUTTO “if - else”

```
if ( condizione )  
    istruzione;
```

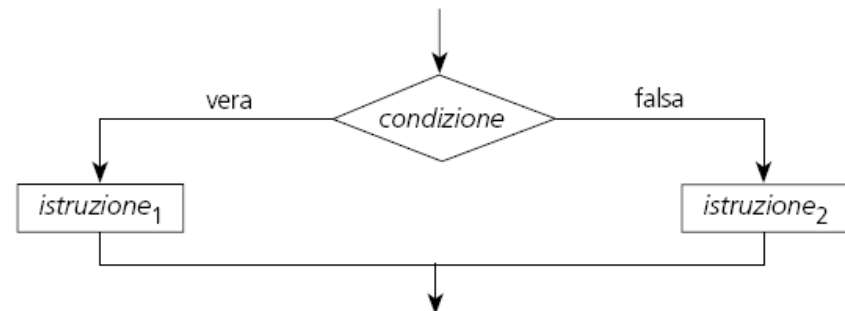
oppure

```
if ( condizione )  
{  
    istruzione;  
    ...  
}
```



oppure

```
if ( condizione )  
{  
    istruzione_1;  
    ...  
}  
else  
{  
    istruzione_2;  
    ...  
}
```



Istruzioni condizionali e iterative

- COSTRUTTO “if - else”

una scelta tra molte

```
if (espressione)
    { <istruzione/i> }
else if (espressione)
    { <istruzione/i> }
else if (espressione)
    { <istruzione/i> }
...
else
    { <istruzione/i> }
```

istruzioni if-else innestate

l'else si riferisce sempre all'if più vicino

<pre>y = 2; if (x < 0) if (x == -1) y = 1; else y = 0; se x = -2 y=0 se x = -1 y=1 se x = 7 y=2</pre>	equivale a	<pre>y = 2; if (x < 0) { if (x == -1) y = 1; else y = 0; }</pre>
--	------------	---

Equivalenza tra TRUE e NON ZERO:

if (x) /* equivale a if (x!=0) */

Istruzioni condizionali e iterative

- COSTRUTTO “switch-case”: una scelta tra molte

```
switch (selettore)
{
    case etichetta_1: istruzione/i_1; break;
    case etichetta_2: istruzione/i_2; break;
    ...
    case etichetta_n: istruzione/i_n; break;
    default: istruzione/i_default; // opzionale
}
```

```
switch (giocata_totocalcio)
{
    case '1': cout << "vittoria in casa" << endl; break;
    case '2': cout << "vittoria fuori casa" << endl; break;
    case 'x': cout << "pareggio" << endl; break;
    default: cout << "non è una giocata corretta" << endl;
}
```

Istruzioni condizionali e iterative

operatore ternario, cioè con 3 operandi:

$\text{espr1} \ ? \ \text{espr2} \ : \ \text{espr3}$

viene prima valutata espr1 : se diverso da zero (quindi vera) viene valutata anche espr2 e il suo valore diventa anche il valore dell'intera espressione; altrimenti viene valutata espr3 ed è il suo il valore dell'intera espressione

Esempio

$x = (y < z) \ ? \ y+3 \ : \ z-2;$

equivale a:

if ($y < z$)

$x = y+3;$

else $x = z-2;$

Istruzioni condizionali e iterative

- COSTRUTTO “while”: esegue una sequenza di istruzioni fintanto che una condizione iniziale rimane TRUE. Ogni ripetizione si chiama **iterazione** del ciclo. Si valuta **prima** la condizione e **poi** si esegue **eventualmente** l’iterazione (se la condizione è TRUE).

while (condizione)

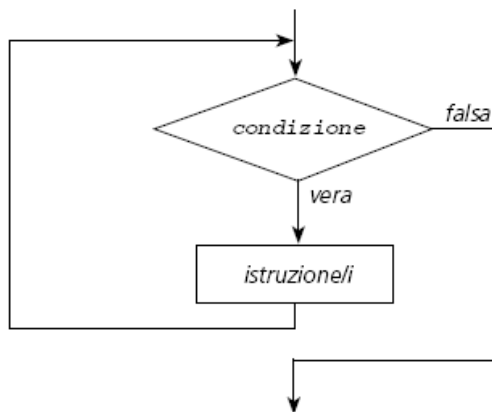
{

statement1;

statement2;

...

}



Esempio

```
int total; int a = 1;
while ( a <= 100) {
    total += a*a;
    a += 1;
}
```

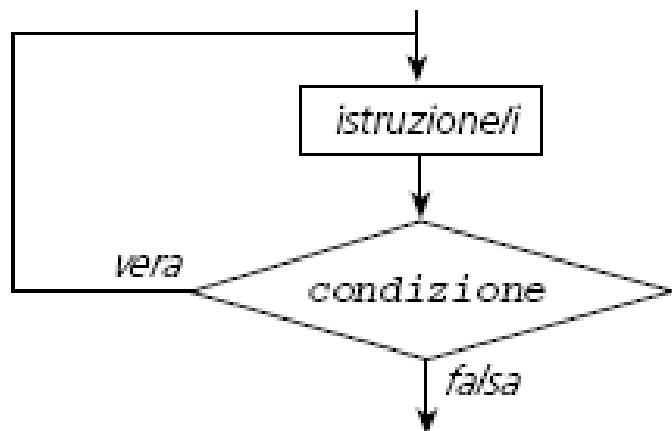
Istruzioni condizionali e iterative

- COSTRUTTO “do while”: la condizione è posta dopo il corpo del ciclo; utile quando si vuole eseguire sempre almeno una volta una iterazione. Poi si valuta la condizione per eventualmente ripetere il ciclo.

do

```
{  
statement1;  
...
```

```
} while (condizione);
```



```
int a = 10;
```

```
do {
```

```
    cout<<"value of a:"<<a<< endl;
```

```
    a = a + 1;
```

```
} while( a < 20 );
```

value of a: 10

value of a: 11

value of a: 12

value of a: 13

value of a: 14

value of a: 15

value of a: 16

value of a: 17

value of a: 18

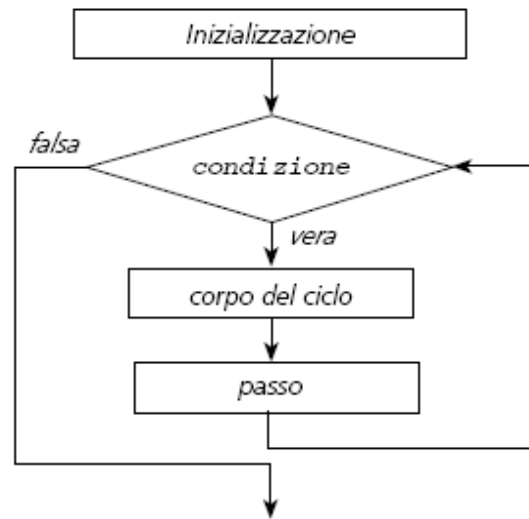
value of a: 19

Istruzioni condizionali e iterative

- COSTRUTTO “for”: utile quando si conosce a priori il numero di iterazioni

```
for (inizializzazione; condizione; passo_del_ciclo)  
  {istruzioni }
```

```
inizializzazione;  
while (condizione)  
{  
  istruzioni;  
  passo_del_ciclo;  
}
```



Esempio

```
int sum , i;  
sum =0;  
for ( i = 1 ; i <= 10; i++)  
{  
  sum += i;  
}
```

Istruzioni condizionali e iterative

- È possibile annidare i cicli uno dentro l'altro

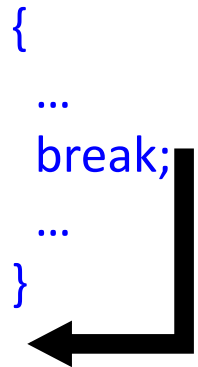
```
do
{
    while (condizione_ciclo_2)
    {
        for (inizializzazione; condizione_ciclo_3; passo)
        {
            ....
        }
    }
} while (condizione_ciclo_1)
```

Istruzioni break e continue

- L'istruzione **break** determina un salto all'istruzione immediatamente successiva al corpo del ciclo o dell'istruzione switch che contengono l'istruzione break. Ovvero consente l'uscita immediata da una istruzione switch, while, for, do-while.

while (condizione)

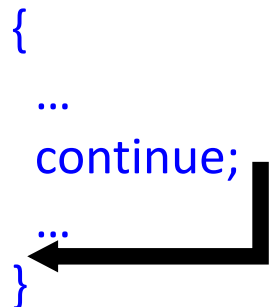
```
{  
  ...  
  break;  
  ...  
}
```



- L'istruzione **continue** provoca la terminazione di un'iterazione del ciclo che la contiene

while (condizione)

```
{  
  ...  
  continue;  
  ...  
}
```



Altri esempi (bool e input/output)

```
#include <iostream>
int main() {
    bool a = true;
    bool b = false;
    bool c = 1;
    if (a || b) std::cout << " OK";
    if (a && b) std::cout << " Impossibile";
    if (a == b) std::cout << " Impossibile";
    if (a != b) std::cout << " OK";
    if (a == true) std::cout << " OK";
    if (a) std::cout << " OK";
    if (!(a || b) && c) std::cout << " Impossibile";
    return 0;
}
```


Enumerazioni

- tipo definito dall'utente costituito da un insieme di costanti

```
enum {enumeratore1, enumeratore2, ..., enumeratore_n};  
enum nome {enumeratore1, ..., enumeratore_n};  
enum nome {  
    enumeratore_1 = espressione_costante_1,  
    ...  
    enumeratore_n = espressione_costante_n, };
```

```
enum {ROSSO, VERDE, AZZURRO};
```

equivalente a:

```
const int ROSSO = 0;
```

```
const int VERDE = 1;
```

```
const int AZZURRO = 2;
```

con il nome:

```
enum giorni_settimana{LUN, MAR, MER, GIO, VEN, SAB, DOM};
```

```
enum giorni_settimana giorno;
```

```
giorno = GIO; //giorno=3
```

Numeri pseudo casuali

- La funzione srand inizializza il generatore di numeri casuali.
La funzione rand() ritorna un numero casuale intero compreso tra 0 e RAND_MAX

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

int main() {
    srand(std::time(nullptr));
    int random_variable = rand();
    std::cout << "RAND_MAX=" << RAND_MAX << std::endl;
    std::cout << "numero casuale tra 0 e RAND_MAX: " << random_variable << std::endl;
    int n = 10;
    std::cout << "numero casuale tra 0 e (n-1): " << rand() % n << std::endl;
    int a = 10;
    int b = 30;
    std::cout << "numero casuale tra a e b: " << a + rand() % (b - a + 1) << std::endl;
}
```

Conversione di tipo

casting: convertire il tipo di una variabile o di un'espressione

- casting implicito: realizzato automaticamente dal compilatore
 - quando si assegna un valore ad una variabile di un altro tipo aritmetico
 - quando si combinano vari tipi nelle espressioni, in questo caso il risultato dell'espressione è automaticamente convertito nel tipo con precisione maggiore
- Esempio: `double d; int i=5;`
`d=i; // il valore di i viene convertito in double`
 - quando si passano argomenti a funzioni che accettano tipi diversi da quelli ricevuti
- casting esplicito: permette al programmatore di impostare esplicitamente la conversione di tipo

In C: **(tipo) espressione_da_convertire**

```
int a = 1; int b = 2; float m = (float)a / (float)b;
```

In C++: **static_cast <tipo> (espressione_da_convertire)**

```
float m = static_cast<float>(a) / static_cast<float>(b);
```

Header files

1) File header di sistema. Esempi:

<code><cstdlib></code> o <code><stdlib.h></code>	General purpose utilities, dynamic memory allocation, random numbers
<code><string></code>	std::basic_string class template
<code><cmath></code> o <code><math.h></code>	Common mathematics functions
<code><iostream></code>	several standard stream objects
<code><ctime></code> o <code><time.h></code>	time/date utilities
<code><fstream></code>	lettura scrittura files

2) File header generati dall'utente. Contengono:

definizione di tipi di dati e classi creati dal programmatore

definizioni di costanti "const"

dichiarazione di prototipi di funzioni

inclusioni e altre direttive

definizione di templates

Operatori di incremento e decremento

++ operatore di incremento $X++$ equivale a $X=X+1$

-- operatore di decremento $X--$ equivale a $X=X-1$

NOTAZIONE PREFISSA (++c) e POSTFISSA (c++) all'interno di espressioni

Gli operatori prefissi modificano il valore della variabile cui sono applicati prima che se ne utilizzi il valore. Gli operatori post-fissi modificano il valore della variabile dopo l'utilizzo del valore (vecchio) nell'espressione.

$x++$ prima usa x nell'istruzione, poi lo incrementa

$++x$ prima incrementa x , poi lo usa nell'istruzione

```
int a = 1, b, c;  
b = a++;          //postincremento; b vale 1 ed a vale 2  
int a = 1, b, c;  
b = ++a;          //preincremento; b vale 2 ed a vale 2
```

```
int a=2, b=3, c; c=++a - b++;
```

Risultato: $c=0$ $a=3$ $b=4$

Equivale a:

```
a = a + 1;
```

```
c=a - b;
```

```
b = b + 1;
```

Operatore divisione e modulo

Un unico operatore di divisione / sia per gli interi che per i reali, ma con significato diverso:

Ad esempio, $10/(-3)$ è una divisione tra interi ed il risultato è -3, mentre $10/3.0$ è una divisione tra reali ed il risultato è 3.333.

Definizione dell'operatore modulo (resto divisione):

$$A\%B = A - (A/B)*B$$

Esempi:

$$1\%4=1, 7\%3=1, 19\%5 = 4$$

Array

- Utilizzati per definire insiemi di variabili omogenee contigue in memoria (vettori e matrici)
- Un array (o vettore) è una sequenza di variabili dello stesso tipo contigue in memoria
- Le variabili si chiamano elementi dell'array e si numerano consecutivamente 0, 1, 2, 3.. ; questi numeri si dicono indici dell'array, ed il loro ruolo è quello di localizzare la posizione di ogni elemento nell'array, fornendo accesso diretto ad esso
- il tipo di elementi immagazzinati nell'array può essere qualsiasi tipo di dato predefinito del C++, ma anche tipi di dato definiti dall'utente
- se il nome del vettore è a, allora a[0] è il nome del primo elemento, a[1] è il nome del secondo elemento, eccetera; l'elemento i-esimo si trova quindi nella posizione i-1, e se l'array ha n elementi, i loro nomi sono a[0], a[1], ... , a[n-1]

a	25.1	34.2	5.25	7.45	6.09	7.54
	0	1	2	3	4	5

Array

DEFINIZIONE (con allocazione statica della memoria): ***tipo nome_variabile[dimensione];***

tipo dichiara il tipo di dato degli elementi che costituiscono l'array

dimensione definisce il numero di elementi, **deve essere una espressione costante**

Esempi:

Tipo di dato dell'array
Nome dell'array
Dimensione dell'array
int numeri [10];
Le parentesi quadre
sono obbligatorie

```
int V[10]; /* variabile V come vettore di 10 int */
```

L'indice assume valori interi compresi tra 0 e ***dimensione-1***, cioè il primo elemento di un vettore di dimensione N è in posizione 0 e l'ultimo è in posizione N-1 (non in posizione N).

Attenzione : l'espressione V[10] non comporta nessun errore (né durante la compilazione né durante l'esecuzione) però V[10] utilizza erroneamente la stessa area di memoria riservata ad altre variabili

Array

E' buona pratica di programmazione utilizzare una costante simbolica per definire la lunghezza di un array:

```
#define N 100 oppure const int N=100;  
int a[N];
```

Inizializzazione di un array: `int V[5] = {1,2,3,4,6};`

Assegnamento del valore 11 all'elemento 3 di V: `V[3] = 11;`

Assegnamento del valore K all'elemento i-esimo di V: `V[i] = K;`

La manipolazione di un vettore avviene elemento per elemento. Esempio: per "copiare" il vettore V1 in V2 con dimensione 4:

```
for (int i=0; i<4; i++) {  
    V2[i] = V1[i];  
    std::cout << "V2[" << i << "] = " << V2[i] << std::endl; }  

```

Errore: `V2 = V1;`

Puntatori

Qualsiasi variabile viene memorizzata a partire da un certo indirizzo in memoria e occupa un certo numero di byte.

Una variabile puntatore contiene in genere l'indirizzo di un'altra variabile che a sua volta contiene un valore.

Le variabili di tipo puntatore si dichiarano come:

```
int *varPtr;
```

la dichiarazione avviene tramite l'uso dell'operatore *

Nota: se vogliamo definire tre puntatori è un errore scrivere:

```
int *var1Ptr, var2Ptr, var3Ptr;
```

infatti in questo modo si dichiara solo var1Ptr come puntatore e var2Ptr e var3Ptr come interi, ovvero il compilatore interpreta la dichiarazione come

```
int *var1Ptr;
```

```
int var2Ptr, var3Ptr;
```

Puntatori

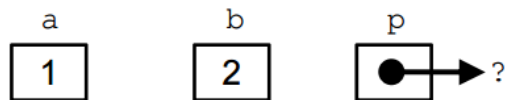
Se v è una variabile, $\&v$ è la locazione o indirizzo di memoria dove è memorizzato il valore di v

Esempio:

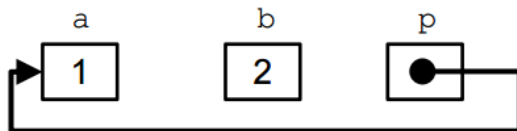
```
int *p; p = NULL; /*puntatore NULLO, non punta a nessun dato*/  
p = &i; /* si dice che "p punta alla variabile i" */
```

Per ottenere il valore contenuto nella cella di memoria dato un indirizzo si usa l'operatore $*$ (operatore di risoluzione del riferimento)

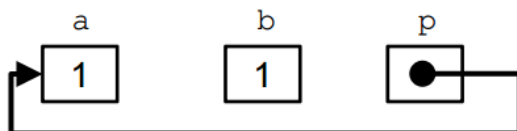
```
int a=1, b=2, *p;
```



```
p=&a;
```



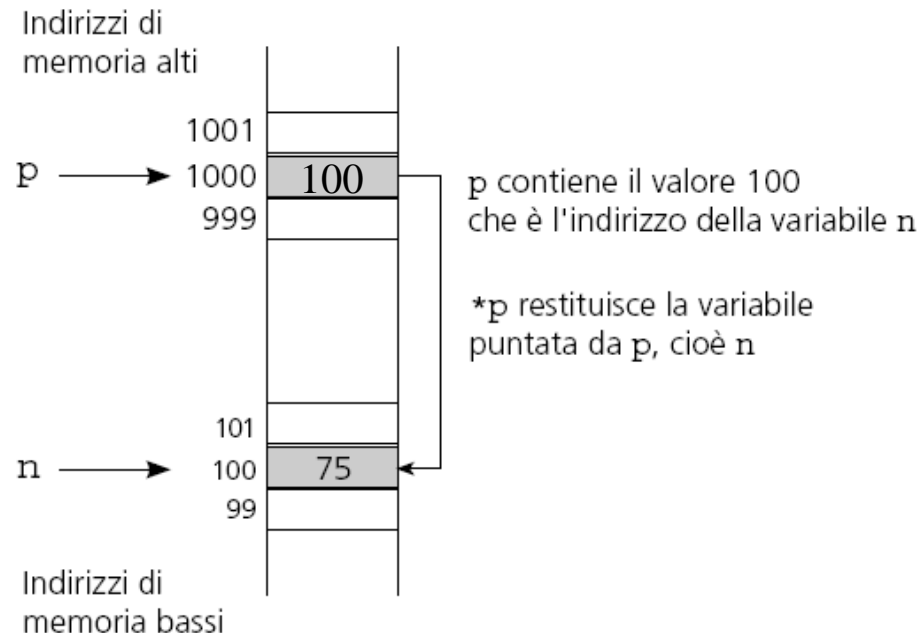
```
b=*p; /* assegno a b il valore puntato da p */
```



Puntatori

puntatori e indirizzi di memoria

```
int n; // supponiamo che n abbia indirizzo 100
int* p; // p è variabile di tipo puntatore ad int
p = &n; // p contiene il valore dell'indirizzo di n
*p = 75; // scrive 75 nella variabile n
```



Puntatori

Altro esempio:

```
int a=17;  
int *p=&a;  
*p=18; // a=18  
*p = *p * 3; // a = 54
```

Costrutti a cui non è possibile puntare:

Costanti: &3; /* illegale */

Espressioni: &(k + 99) /* illegale */

Un puntatore può essere inizializzato a NULL (0 equivale a **NULL** o **nullptr**):

```
int *p= NULL;
```

E' possibile operare un confronto tra un puntatore e NULL

```
if (p != NULL) .....
```

Puntatori

scrivendo `*(p + 3)` l'indirizzo base del puntatore `p` viene incrementato automaticamente dell'equivalente di 3 locazioni del tipo a cui punta `p` (si chiama **aritmetica dei puntatori**)

Esempio:

```
#define N 5
```

```
int a[N] = { 11, 3, 71, 44, 5 };
```

```
int *p= a;
```

```
std::cout << *(p + 3); // stampa 44
```

Se il sistema assegna all'indirizzo base dell'array `a` l'indirizzo 300, l'elemento `a[0]` è all'indirizzo 300, `a[1]` a 304, `a[2]` a 308, ecc. (supponendo che la macchina usata riservi 4 byte per un `int`)

il nome di una variabile array è un puntatore *costante* al primo elemento dell'array

Attenzione: essendo i nomi degli array puntatori costanti

`a=p`; `a++`; `a+=2`; sono espressioni illegali; il valore di `a` non può essere modificato.

Puntatori

L'operatore di indirizzo & può essere applicato sia a variabili che a elementi di array

```
int a[10];
```

```
&a[3] /* lecita, equivale a &a+3*sizeof(int) */
```

Altro esempio:

```
int a[5]={10,20,30,40,50};
```

```
int * ptr1=a;
```

```
int * ptr2=&a[0]; //ptr1,ptr2 e a puntano alla stessa cella
```

```
int val1=*(ptr1 + 1); //val1=20
```

```
int val2=ptr1[3]; //val2=40
```

I puntatori hanno alcuni usi fondamentali:

- Allocazione dinamica della memoria
- Argomento funzioni che devono modificare le variabili passate
- Strutture dati dinamiche

Allocazione dinamica della memoria

Quando non è possibile prevedere la dimensione della memoria richiesta in un programma al momento della compilazione (per esempio quando vogliamo costruire un array la cui dimensione viene inserita da tastiera dall'utente) si utilizza il metodo di allocazione dinamica.

In C++ per realizzare l'allocazione dinamica si utilizzano gli operatori **new** e **delete**

- Per allocare dinamicamente una variabile si utilizza **new**
- E' necessario specificare il tipo della variabile da allocare
- L'operatore **new** restituisce l'indirizzo dell'area di memoria allocata e deve essere assegnato ad un puntatore

NomeTipo *var_ptr;

var_ptr= new NomeTipo; new crea una variabile della dimensione appropriata

La memoria allocata dinamicamente ha un tempo di vita che non è legato al tempo di esecuzione del blocco di codice in cui essa viene creata, nel senso che può sopravvivere, se non viene deallocata, anche dopo la conclusione del blocco di codice

Allocazione dinamica della memoria

Per liberare la memoria allocata dinamicamente di variabili di cui non si richiede ulteriore impiego si utilizza l'operatore **delete** applicato ad un puntatore

NomeTipo *var_ptr;

var_ptr= new NomeTipo; new crea una variabile della dimensione appropriata

.....

delete var_ptr; delete dealloca (libera) l'area di memoria puntata dal puntatore

new può inizializzare un oggetto e ne invoca il costruttore (vedremo in seguito)

double * var_ptr=new double(3.1415);

Allocazione dinamica della memoria

Allocazione dinamica di un array (in questo caso la dimensione può essere una variabile):

```
int dim=5;
```

```
NomeTipo *array_ptr;
```

```
array_ptr= new NomeTipo[dim];
```

```
...
```

```
array_ptr[3]=....
```

```
...
```

```
delete [] array_ptr;
```

Il rilascio (liberazione) della memoria deve essere fatto esplicitamente usando la funzione delete

Non si può inizializzare un array allocato dinamicamente

```
double * array_ptr=new double[100] (3.14) !!!ERRORE
```

Allocazione dinamica della memoria

L'uso improprio dei puntatori e dell'allocazione dinamica della memoria porta ad errori a tempo di esecuzione del programma (segmentation fault) o a falle nella memoria (memory leak)

```
int ia[10];
```

```
ia[10] = 4; // possibile segmentation fault
```

```
int* p1; //No inizializzazione, un puntatore dovrebbe essere sempre inizializzato!
```

```
*p1 = 3; //possibile segfault
```

```
void myfunc(...) // memory leak
```

```
{
```

```
    int *p=new int[100];
```

```
    ... return;
```

```
}
```

Puntatori a puntatori

Una variabile puntatore è a sua volta memorizzata in memoria, quindi ha un indirizzo!

Posso memorizzarlo? Sì → puntatori a puntatori

Esempi: `int **doppiopunt;`

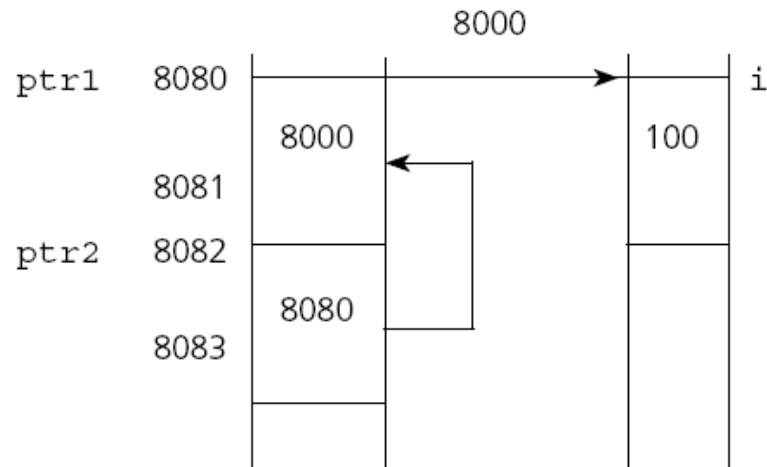
```
int i = 100;
```

```
int* ptr1 = &i;
```

```
int** ptr2 = &ptr1;
```

`ptr1` è puntatore ad intero e punta la variabile `i` di tipo `int`;

`ptr2` è puntatore a un puntatore di interi e punta alla variabile `ptr1`



Puntatori a puntatori

Una variabile puntatore è a sua volta memorizzata in memoria, quindi ha un indirizzo!

Posso memorizzarlo? Sì → puntatori a puntatori

Esempio:

```
int **doppiopunt;
```

A che cosa servono i puntatori a puntatori?

- Allocazione dinamica di array multidimensionali
- Passaggio a funzioni, gestione parametri riga di comando

```
#include <string>
int main(int argc, const char **argv)
{
    std::string word = argv[1]; //word contiene il primo parametro
    return 0;
}
```

Puntatori costanti e puntatori a costanti

- Puntatore costante:

```
<tipo_dato>* const <nome_puntatore> = <indirizzo_variabile>;
```

```
int x, e;
```

```
int* const p = &x;
```

```
*p = e; // corretto
```

```
p = &e // scorretto; p non può cambiare valore
```

- Puntatore ad una costante:

```
const <tipo_dato>* <nome_puntatore> = <indirizzo_const>;
```

```
const int x = 25;
```

```
const int e = 50;
```

```
const int* p = &x;
```

```
*p = 15; // scorretto; ciò che p punta non può cambiare valore
```

```
p = &e // corretto
```

Array bidimensionali

- gli array bidimensionali (matrici) hanno due dimensioni e, pertanto, due indici; la sintassi per la dichiarazione è:

tipo_elemento nome_array [NumRighe][NumColonne]

- inizializzazione:

```
int tabella[2][3] = {{51, 52, 53}, {54, 55, 56}};
```

$$\begin{bmatrix} 51 & 52 & 53 \\ 54 & 55 & 56 \end{bmatrix}$$

- assegnamenti:

```
int x = tabella[1][0]; // assegna ad x 54  
tabella[1][2] = 58; // sostituisce 56 a 58
```

Array bidimensionali

Esempio: inizializzazione di una tabella con tre righe e quattro colonne di numeri interi:

```
int a[3][4] = {
{0, 1, 2, 3} , /* initializers for row indexed by 0 */
{4, 5, 6, 7} , /* initializers for row indexed by 1 */
{8, 9, 10, 11} /* initializers for row indexed by 2 */
};
```

L'elemento di indice (i,j) si scrive a[i][j], osservazione: allocazione dinamica non banale

```
#include <iostream>
int main () {
int a[5][2] = {{0,0}, {1,2}, {2,4}, {3,6},{4,8}};
for ( int i = 0; i < 5; i++ )
    for ( int j = 0; j < 2; j++ ) {
        std::cout << "a[" << i << "]["<< j << "]: ";
        std::cout << a[i][j]<< std::endl; }
return 0; }
```