Corso “Programmazione 1”
Capitolo 02: Variabili, Costanti, Tipi

Docente: Marco Roveri - marco.roveri@unitn.it
Esercitatori: Giovanni De Toni - giovanni.detoni@unitn.it
Stefano Berlato - stefano.berlato-1@unitn.it

C.D.L.: Informatica (INF)
A.A.: 2021-2022
Luogo: DISI, Università di Trento
URL: https://bit.ly/2VgfYwJ

Ultimo aggiornamento: 22 settembre 2021

Terms of Use and Copyright

The material (text, figures) in these slides is authored mostly by Roberto Sebastiani, with contributions by Marco Roveri, Alessandro Armando, Enrico Giunchiglia e Sabrina Recla.

Operatore di Assegnazione

- Sintassi dell'operatore di assegnazione: exp1 = exp2
  - exp1 deve essere un'espressione dotata di indirizzo (l-value)
  - exp1 e exp2 devono essere di tipo compatibile
- Il valore di *exp2* viene valutato e poi assegnato a *exp1*
  Esempio di assegnazioni:
  { ESEMPI_BASE/assegnazione_errori.cc }
- Un'assegnazione può occorrere dentro un'altra espressione.
  - Il valore di un'espressione di assegnazione è il valore di exp2.
  - L'operazione di assegnazione, '=', associa a destra.
- Esempio:
  ```
  a = b = c = d = 5;
  ```
  è equivalente a:
  ```
  (a=(b=(c=(d=5))));
  ```
  Esempio di assegnazioni multiple:
  { ESEMPI_BASE/assegnazione.cc }

Operatori misti assegnazione/aritmetica

Forma compatta	Forma estesa
`x += y`	`x = x+y`
`x -= y`	`x = x-y`
`x *= y`	`x = x*y`
`x /= y`	`x = x/y`
`x %= y`	`x = x%y`

L'uso della forma compatta è tipicamente più efficiente
(su alcune architetture consente di utilizzare in modo ottimale funzionalità della CPU)

Esempi di operatori di assegnazione misti:
{ `ESEMPI_BASE/op_assegnazione.cc` }

Operatori di incremento e decremento unitario

- `x++`:
  - incrementa `x` di un'unità
  - denota il valore di `x` prima dell'incremento
- `x--`:
  - decrementa `x` di un'unità
  - denota il valore di `x` prima del decremento
- `++x`:
  - incrementa `x` di un'unità
  - denota il valore di `x` dopo l'incremento
- `--x`:
  - decrementa `x` di un'unità
  - denota il valore di `x` dopo il decremento

Nota

L'uso della forma compatta: "`x++;`" è tipicamente più efficiente della forma estesa corrispondente: "`x = x + 1;`"

Esempi di operatori di incremento unitario:
{ `ESEMPI_BASE/op_incremento.cc` }

Operatori Relazionali

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

Ordine di valutazione di un'espressione

- In C++ *non* è specificato l'ordine di valutazione degli operandi di ogni operatore, in particolare:
  - l'ordine di valutazione di sottoespressioni in un'espressione
  - (l'ordine di valutazione degli argomenti di una funzione)
- Es: nel valutare `expr1 <op> expr2`, non è specificato se `expr1` venga valutata prima di `expr2` o viceversa
  (Con alcune importanti eccezioni, vedi operatori Booleani)
- Problematico quando sotto-espressioni contengono operatori con "side-effects" (e.g. gli operatori di incremento). Esempio:

```
j = i++ * i++;    // undefined behavior
i = ++i + i++;    // undefined behavior
```

  $\Longrightarrow$ evitare l'uso di operatori con side-effects in sotto-espressioni

Per approfondimenti si veda ad esempio
`http://en.cppreference.com/w/cpp/language/eval_order`

Il tipo Booleano

Il tipo Booleano

Il C++ prevede un tipo Booleano `bool`:

- il valore falso è rappresentato dalla costante `false` (equivalente a 0)
- il valore vero è rappresentato dalla costante `true` (equivalente ad un valore intero diverso da 0)
- si può usare a tal scopo anche il tipo `int`
- operatori Booleani: `!` (not), `&&` (and), `||` (or)

x	y	!x	&&	||
false	false	true	false	false
false	true	true	false	true
true	false	false	false	true
true	true	false	true	true

- `!!x` $\Longleftrightarrow$ `x`,
  `!(x || y)` $\Longleftrightarrow$ `(!x && !y)`,
  `!(x && y)` $\Longleftrightarrow$ `(!x || !y)`,

Il tipo Booleano (II)

Nota 1: Priorità degli operatori Booleani

- l'operatore `!` ha priorità sugli operatori `&&` e `||`:
  "`!x && y`" è equivalente a "`(!x) && y`", non a "`!(x && y)`"

Nota 2: Lazy Evaluation

- In C++ `&&` e `||` sono valutati in modi "pigro" (lazy evaluation):
  - in `(exp1 && exp2)`, se `exp1` è valutata `false`, allora `exp2` non viene valutata
  - idem se `exp1` è valutata `true` in `(exp1 || exp2)`
  - l'ordine di valutazione tra `exp1` e `exp2` è da sinistra a destra

$\Longrightarrow$ più efficiente

$\Longrightarrow$ può causare seri effetti collaterali se usata con espressioni che modificano valori di variabili (es "++")

$\Longrightarrow$ evitare di usare tali costrutti all'interno di espressioni Booleane

Il tipo Booleano (esempi)

Esempio di valori Booleani rappresentati con bool:
{ `ESEMPI_BASE/booleano_bool.cc` }
Esempio di lazy evaluation con operatori Booleani:
{ `ESEMPI_BASE/booleano_sideeffects.cc` }

I Tipi Reali

- I tipi reali hanno come insieme di valori un sottoinsieme dei numeri reali, ovvero quelli rappresentabili all'interno del computer in un formato prefissato
- Ne esistono vari tipi, in ordine crescente di precisione:
  - i `float`
  - i `double`
  - i `long double`
- Operatori aritmetici: `+`, `-`, `*`, `/`
  ("/" diverso da divisione tra interi: `7.0/2.0 = 3.5`)
- Precisione e occupazione di memoria dipendono dalla macchina

Nota

- in realtà sottoinsieme dei razionali
- anche i tipi reali hanno un range finito!

I Tipi Reali: esempi

```
double a = 2.2, b = -14.12e-2;
double c = .57, d = 6.;

float g = -3.4F;  //literal float
float h = g-.89F; //suffisso F (f)

long double i = +0.001;
long double j = 1.23e+12L;
// literal long double
// suffisso L (l)
```

Esempi con i tipi reali:
{ ESEMPI_BASE/reali.cc }
Esempi di confronto tra tipi reali e interi:
{ ESEMPI_BASE/reali_vs_interi.cc }

Precisione dei tipi reali

- La rappresentazione dei numeri reali ha intrinsici limiti di precisione, dovuti a:
  - limitato numero di bit nella rappresentazione della mantissa (vedi codifica floating-point)
  - uso di codifica binaria nei decimali
    Es: "`.100110011001`" significa "$1 \cdot \frac{1}{2} + 0 \cdot \frac{1}{4} + 0 \cdot \frac{1}{8} + 1 \cdot \frac{1}{16} + ...$"
    $\Longrightarrow$ alcuni numeri non hanno rappresentazione esatta
    (Es: 0.1, 11.1,...):

$\Longrightarrow$ Intrinseca sorgente di errori di precisione

- talvolta non visualizzabili con "`cout << ...`";
- confronto con "`... == ...`" tra tipi reali spesso problematico

Problemi di precisione:

{ `ESEMPI_BASE/reali_precisione.cc` }

Il Tipo Enumerato

- Un tipo enumerato è un insieme finito di costanti intere, definite dal programmatore, ciascuna individuata da un identificatore
- Sintassi: `enum typeid { id_or_init1, ..., id_or_initn }`
- Se non specificato esplicitamente, i valori sono equivalenti rispettivamente agli interi `0, 1, 2, ...`
- Ad una variabile di tipo enumerativo è possibile assegnare solo un valore del tipo enumerativo
- I valori vengono stampati come interi!

Esempi di uso di enumerativi:
{ `ESEMPI_BASE/enum.cc` }

Il tipo Carattere I

- Il tipo `char` ha come insieme di valori i caratteri stampabili
  - es. `'a', 'Y', '6', '+', ' '`
  - generalmente un carattere occupa 1 byte (8 bit)
- Il tipo `char` è un sottoinsieme del tipo `int`
- Il valore numerico associato ad un carattere è detto codice e dipende dalla codifica utilizzata dal computer
  - es. ASCII, EBCDIC, BCD, ...
  - la più usata è la codifica ASCII

Il tipo Carattere II

Nota importante:

Il tipo `char` è indipendente dalla particolare codifica adottata!

$\Longrightarrow$ un programma deve funzionare sempre nello stesso modo, indipendentemente dalla codifica usata nella macchine su cui è eseguito!!

$\Longrightarrow$ evitare di far riferimento al valore ASCII di un carattere:

```
char c;
c = 65;  // NO!!!!!!
c = 'A'; // SI
```

Codifica dei caratteri: regole generali

Qualunque codifica deve soddisfare le seguenti regole

- Precedenza:
  - `'a' < 'b' < ... < 'z'`
  - `'A' < 'B' < ... < 'Z'`
  - `'0' < '1' < ... < '9'`
- La consecutività tra lettere minuscole, lettere maiuscole, numeri
  - `'a', 'b', ..., 'z'`
  - `'A', 'B', ..., 'Z'`
  - `'0', '1', ..., '9'`

Nota

Non è fissa la relazione tra maiuscole e minuscole o fra i caratteri non alfabetici

La codifica ASCII

0	NUL	1	^A	2	^B	3	^C	4	^D	5	^E	6	^F	7	^G
8	^H	9	^I	10	^J	11	^K	12	^L	13	^M	14	^N	15	^O
16	^P	17	^Q	18	^R	19	^S	20	^T	21	^U	22	^V	23	^W
24	^X	25	^Y	26	^Z	27	^[	28	^\	29	^]	30	^^	31	^-
32	SP	33	!	34	"	35	#	36	$	37	%	38	&	39	'
40	(	41	)	42	*	43	+	44	,	45	-	46	.	47	/
48	0	49	1	50	2	51	3	52	4	53	5	54	6	55	7
56	8	57	9	58	:	59	;	60	<	61	=	62	>	63	?
64	@	65	A	66	B	67	C	68	D	69	E	70	F	71	G
72	H	73	I	74	J	75	K	76	L	77	M	78	N	79	O
80	P	81	Q	82	R	83	S	84	T	85	U	86	V	87	W
88	X	89	Y	90	Z	91	[	92	\	93	]	94	^	95	_
96	`	97	a	98	b	99	c	100	d	101	e	102	f	103	g
104	h	105	i	106	j	107	k	108	l	109	m	110	n	111	o
112	p	113	q	114	r	115	s	116	t	117	u	118	v	119	w
120	x	121	y	122	z	123	{	124	|	125	}	126	~	127	DEL

Esempio sull'uso dei caratteri

```
char c ='f';
n ='\n';
char l = 'a';
((l >= 'a')&&(l <= 'z')) // test: l e' una
                         // lettera minuscola?
l += 3;      // l diventa 'd'
l--;         // l diventa 'c'
l-='a'-'A'; // l diventa 'C'
```

Nota

È possibile applicare operatori aritmetici agli oggetti di tipo `char`!

Esempio sull'uso di `char`:
{ `ESEMPI_BASE/char.cc` }

L'Operatore `sizeof`

- L'operatore `sizeof`, può essere prefisso a:
  - una variabile (esempio: `sizeof (x)`)
  - una costante (esempio: `sizeof ('a')`)
  - al nome di un tipo (esempio: `sizeof (double)`)
- Può essere usato in alcuni casi con o senza parentesi
  - `sizeof x` Ok
  - `sizeof double` Ko
- Restituisce un intero rappresentante la dimensione in byte
- È applicabile a espressioni di qualsiasi tipo (non solo ai tipi fondamentali)

Esempio di uso di `sizeof` applicato a tipi:
{ `ESEMPI_BASE/sizeof.cc` }
Esempio di uso di `sizeof` applicato a espressioni:
{ `ESEMPI_BASE/sizeof2.cc` }

Operazioni miste e conversioni di tipo

- Spesso si usano operandi di tipo diverso in una stessa espressione o si assegna ad una variabile un valore di tipo diverso della variabile stessa
- In ogni operazione mista è sempre necessaria una conversione di tipo che può essere
  - implicita
  - esplicita

Esempio:

```
int prezzo = 27500;
double peso = 0.3;
int costo = prezzo * peso;
```

Conversioni Implicite

- Le conversioni implicite vengono effettuate dal compilatore
- Le conversioni implicite più significative sono:
  - nella valutazione di espressioni numeriche, gli operandi sono convertiti al tipo di quello di dimensione maggiore
  - nell'assegnazione, un'espressione viene sempre convertita al tipo della variabile

Esempi:

```
float x = 3;    //equivale a: x = 3.0
int y = 2*3.6; //equivale a: y = 7
```

Esempio di conversioni implicite:
{ ESEMPI_BASE/conversioni_err.cc }
Esempio di conversioni implicite:
{ ESEMPI_BASE/conversioni_corr.cc }

Conversioni Esplicite

- Il programmatore può richiedere una conversione esplicita di un valore da un tipo ad un altro (casting)
- Esistono due notazioni:
  - prefissa. Esempio:
    ```
    int i = (int) 3.14;
    ```
  - funzionale. Esempio:
    ```
    double f = double(3)
    ```

Conversioni tra tipi numerici - approfondimenti

- Promozione: conversione da un tipo ad uno simile più grande
  - `short` $\Longrightarrow$ `int` $\Longrightarrow$ `long` $\Longrightarrow$ `long long`,
    `float` $\Longrightarrow$ `double` $\Longrightarrow$ `long double`
  - garantisce di mantenere lo stesso valore
- Conversioni tra tipi compatibili
  - Conversione da tipo reale a tipo intero: il valore viene troncato
    `int x = (int)  4.7` $\Longrightarrow$ `x==4`,
    `int x = (int) -4.7` $\Longrightarrow$ `x==-4`
  - Conversione da tipo intero a reale: il valore può perdere precisione
    `float y = float(2147483600); // 2^31-48`
    $\Longrightarrow$ `y == 2147483648.0; // 2^31`

Esempio di conversioni tra tipi numerici:
{ `ESEMPI_BASE/conversioni_miste.cc` }
Esempio di conversioni tra tipi reali:
{ `ESEMPI_BASE/conversioni_real.cc` }

Per approfondimenti vedere, ad esempio:
`http://www.cplusplus.com/doc/tutorial/typecasting/`