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

Docente:	Roberto Sebastiani - roberto.sebastiani@unitn.it
Esercitatori:	Mario Passamani - mario.passamani@unitn.it
	Alessandro Tomasi - alessandro.tomasi@unitn.it

C.D.L.:	Informatica (INF)
	Ing. Informatica, delle Comunicazioni ed Elettronica (ICE)
	Studenti con numero di matricola pari
A.A.:	2019-2020
Luogo:	DISI, Università di Trento
URL:	disi.unitn.it/rseba/DIDATTICA/prog1_2020/

Ultimo aggiornamento: 30 ottobre 2019, ore 11:29

Outline

Le Variabili e le Costanti

- Per memorizzare un valore in un'area di memoria si utilizzano entità chiamate variabili o costanti.
- Le variabili permettono la modifica del loro valore durante l'esecuzione del programma.
- L' area di memoria corrispondente è identificata da un nome, che ne individua l'indirizzo in memoria.

Le Variabili

Le variabili sono caratterizzate da una quadrupla:

- il nome (identificatore)
- il tipo
- il locazione di memoria (l-value o indirizzo)
- il valore (r-value)

Definizione e Dichiarazione di Variabili I

- Definizione:
  - Formato: `tipo identificatore;`
  - Esempio: `int x;`
  - Se (e solo se!) una variabile è definita esternamente ad ogni funzione, `main()` inclusa, (variabile globale) viene automaticamente inizializzata al valore 0
- Definizione con inizializzazione:
  - Formato: `tipo identificatore=exp;`
  - Esempio: `int x=3*2;`
  - Esempio: `int y=3*z; // z definita precedentemente`
- Dichiarazione:
  - Formato: `extern tipo identificatore;`
  - Esempio: `extern int x;`

output di variabile

Per produrre in output il valore di una variabile x, si usa l'istruzione:

```
cout << x ...
```

Definizione e Dichiarazione di Variabili I

- Definizione:
  - Formato: `tipo identificatore;`
  - Esempio: `int x;`
  - Se (e solo se!) una variabile è definita esternamente ad ogni funzione, `main()` inclusa, (variabile globale) viene automaticamente inizializzata al valore 0
- Definizione con inizializzazione:
  - Formato: `tipo identificatore=exp;`
  - Esempio: `int x=3*2;`
  - Esempio: `int y=3*z; // z definita precedentemente`
- Dichiarazione:
  - Formato: `extern tipo identificatore;`
  - Esempio: `extern int x;`

output di variabile

Per produrre in output il valore di una variabile x, si usa l'istruzione:

```
cout << x ...
```

Definizione e Dichiarazione di Variabili I

- Definizione:
  - Formato: `tipo identificatore;`
  - Esempio: `int x;`
  - Se (e solo se!) una variabile è definita esternamente ad ogni funzione, `main()` inclusa, (variabile globale) viene automaticamente inizializzata al valore 0
- Definizione con inizializzazione:
  - Formato: `tipo identificatore=exp;`
  - Esempio: `int x=3*2;`
  - Esempio: `int y=3*z; // z definita precedentemente`
- Dichiarazione:
  - Formato: `extern tipo identificatore;`
  - Esempio: `extern int x;`

output di variabile

Per produrre in output il valore di una variabile x, si usa l'istruzione:

`cout << x ...`

Definizione e Dichiarazione di Variabili I

- Definizione:
  - Formato: `tipo identificatore;`
  - Esempio: `int x;`
  - Se (e solo se!) una variabile è definita esternamente ad ogni funzione, `main()` inclusa, (variabile globale) viene automaticamente inizializzata al valore 0
- Definizione con inizializzazione:
  - Formato: `tipo identificatore=exp;`
  - Esempio: `int x=3*2;`
  - Esempio: `int y=3*z; // z definita precedentemente`
- Dichiarazione:
  - Formato: `extern tipo identificatore;`
  - Esempio: `extern int x;`

output di variabile

Per produrre in output il valore di una variabile x, si usa l'istruzione:

```
cout << x ...
```

Definizione e Dichiarazione di Variabili II

- Definizione: quando il compilatore incontra una definizione di una variabile, esso predispone l'allocazione di un'area di memoria in grado di contenere la variabile del tipo scelto
- Dichiarazione: specifica solo il tipo della variabile e presuppone dunque che la variabile venga definita in un'altra parte del programma

Esempio di definizioni di variabili:
{ `ESEMPI_BASE/variabili.cc` }
... con inizializzazione:
{ `ESEMPI_BASE/variabili2.cc` }

Dichiarazione di Costanti

- Sintassi:
  - Formato: `const tipo identificatore = exp;`
  - `exp` deve essere una espressione il cui valore deve poter essere calcolato in fase di compilazione
    (su alcuni compilatori è possibile inizializzare costanti a valore non costanti, ma il risultato è imprevedibile e varia a seconda dei casi $\Longrightarrow$ da evitare assolutamente)
- Esempi

```
const int kilo = 1024;
const double pi = 3.14159;
const int mille = kilo - 24;
```

Esempio di uso di costanti:
{ ESEMPI_BASE/costanti.cc }

Outline

Concetto di stream

- Un programma comunica con l'esterno tramite uno o più flussi di caratteri (stream)
- Uno stream è una struttura logica costituita da una sequenza di caratteri, in numero teoricamente infinito, terminante con un apposito carattere che ne identifica la fine.
- Gli stream vengono associati (con opportuni comandi) ai dispositivi fisici collegati al computer (tastiera, video) o a file residenti sulla memoria di massa

Stream predefiniti

- In C++ esistono i seguenti stream predefiniti
  - cin (stream standard di ingresso)
  - cout (stream standard di uscita)
  - cerr (stream standard di errore)
- Le funzioni che operano su questi stream sono in una libreria di ingresso/uscita e per usarle occorre la direttiva:

```
#include <iostream>
```

Stream di uscita

- Lo stream di uscita standard (cout) è quello su cui il programma scrive i dati
  - è tipicamente associato allo schermo
- Per scrivere dati si usa l'istruzione di scrittura:

```
stream << espressione;
```

- L'istruzione di scrittura comporta:
  - il calcolo del valore dell'espressione
  - la conversione in una sequenza di caratteri
  - il trasferimento della sequenza nello stream

Nota:

La costante predefinita `endl` corrisponde ad uno `'\n'`, per cui viene iniziata una nuova linea con il cursore nella prima colonna

Stream di uscita

- Lo stream di uscita standard (cout) è quello su cui il programma scrive i dati
  - è tipicamente associato allo schermo
- Per scrivere dati si usa l'istruzione di scrittura:

  ```
  stream << espressione;
  ```
- L'istruzione di scrittura comporta:
  - il calcolo del valore dell'espressione
  - la conversione in una sequenza di caratteri
  - il trasferimento della sequenza nello stream

Nota:

La costante predefinita `endl` corrisponde ad uno `'\n'`, per cui viene iniziata una nuova linea con il cursore nella prima colonna

Scritture multiple

- L'istruzione di scrittura ha una forma più generale che consente scritture multiple, nel formato

```
cout << espressione1 << espressione2 << ...
```

- Ad esempio,

```
cout << x << y << endl;
```

corrisponde a:

```
cout << x;
cout << y;
cout << endl;
```

Esempio di uso di operazioni di output:
{ ESEMPI_BASE/esempio_cout.cc }
... con output multiplo:
{ ESEMPI_BASE/esempio_cout_multiplo.cc }

Stream di ingresso

- Lo stream di ingresso standard (cin) è quello da cui il programma preleva i dati
  - è tipicamente associato alla tastiera
- Per prelevare dati si usa l'istruzione di lettura:

  `stream >> espressione;`

  dove `espressione` deve essere un'espressione dotata di indirizzo (per ora, una variabile)
- L'istruzione di lettura comporta in ordine:
  1. il prelievo dallo stream di una sequenza di caratteri
  2. la conversione di tale sequenza in un valore che viene assegnato alla variabile

Lettura di un carattere da cin

- Consideriamo l'istruzione `cin >> x`, dove `x` è di tipo `char`

  □□□□*a*□□-14.53□□728□□ ...
  ⇑

  (qui "□" rappresenta uno spazio e "⇑" il cursore)
- se il carattere puntato dal cursore è una spaziatura:
  - il cursore si sposta in avanti per trovare un carattere che non sia una spaziatura
- se il carattere puntato dal cursore non è una spaziatura:
  1. il carattere viene prelevato
  2. il carattere viene assegnato alla variabile x
  3. il puntatore si sposta alla casella successiva

Lettura di un numero da cin

- Consideriamo l'istruzione `cin >> x`, dove `x` è un numero (ad esempio di tipo `int`)
- se il carattere puntato dal cursore è una spaziatura:
  - come nel caso precedente
- se la sequenza di caratteri è interpretabile come un numero compatibile con il tipo di `x`:

  `□□□□a□□-14.53□□728□□ ...`
  `⇑`

  1. la sequenza di caratteri viene prelevata
  2. la sequenza viene convertita nel suo corrispondente valore (es, il valore intero `728`) che viene assegnato alla variabile x
  3. il puntatore si sposta alla casella successiva
- altrimenti, l'operazione di prelievo non avviene e lo stream si porta in uno stato di errore

Letture multiple

- L'istruzione di ingresso ha una forma più generale che consente letture multiple, nel formato

  `cin >> var1 >> var2 >> ...`

- Ad esempio, se `x`, `y` , `z` sono risp. `char`, `double` e `int`:

  `cin >> x >> y >> z;`

  corrisponde a:

```
cin >> x;
cin >> y;
cin >> z;
```

□□□□*a*□□-14.53□□728□□ ...
⇑

Esempi

Esempio di uso di operazioni di input:
{ `ESEMPI_BASE/esempio_cin.cc` }
... con input multiplo:
{ `ESEMPI_BASE/esempio_cin2.cc` }
... prima intero e poi reale:
{ `ESEMPI_BASE/esempio_cin3.cc` }

Alcune funzioni utili della libreria `<iostream>`

- `cin.eof()`: ritorna un valore diverso da 0 se lo stream cin ha raggiunto la sua fine (End Of File)
  - va usato sempre dopo almeno un'operazione di lettura
  - richiede un separatore dopo l'ultimo elemento letto
- `cin.fail()`: ritorna un valore diverso da 0 se lo stream cin ha rilevato uno stato di errore (e.g. stringa per `int`) o un end-of-file
  - non necessariamente usato dopo almeno un'operazione di lettura
  - non richiede un separatore dopo l'ultimo elemento letto
- `cin.clear()`: ripristina lo stato normale dallo stato di errore

Alcune funzioni utili della libreria `<iostream>`

- `cin.eof()`: ritorna un valore diverso da 0 se lo stream cin ha raggiunto la sua fine (End Of File)
  - va usato sempre dopo almeno un'operazione di lettura
  - richiede un separatore dopo l'ultimo elemento letto
- `cin.fail()`: ritorna un valore diverso da 0 se lo stream cin ha rilevato uno stato di errore (e.g. stringa per `int`) o un end-of-file
  - non necessariamente usato dopo almeno un'operazione di lettura
  - non richiede un separatore dopo l'ultimo elemento letto
- `cin.clear()`: ripristina lo stato normale dallo stato di errore

Alcune funzioni utili della libreria `<iostream>`

- `cin.eof()`: ritorna un valore diverso da 0 se lo stream cin ha raggiunto la sua fine (End Of File)
  - va usato sempre dopo almeno un'operazione di lettura
  - richiede un separatore dopo l'ultimo elemento letto
- `cin.fail()`: ritorna un valore diverso da 0 se lo stream cin ha rilevato uno stato di errore (e.g. stringa per `int`) o un end-of-file
  - non necessariamente usato dopo almeno un'operazione di lettura
  - non richiede un separatore dopo l'ultimo elemento letto
- `cin.clear()`: ripristina lo stato normale dallo stato di errore

Outline

Rappresentazione binaria dei numeri

- La rappresentazione binaria dei numeri avviene tramite sequenze di bit (uni e zeri).
- Distinguiamo la rappresentazione per:
  - numeri interi positivi
  - numeri interi con segno
  - numeri reali

Numeri interi positivi

- Una sequenza di bit $b_{n-1}...b_1 b_0$ rappresenta il numero:

$$\sum_{i=0}^{n-1} 2^i \cdot b_i = 2^{n-1} \cdot b_{n-1} + ... + 4 \cdot b_2 + 2 \cdot b_1 + 1 \cdot b_0$$

- Dati n bit è possibile rappresentare numeri nell'intervallo $[0, 2^n - 1]$ (ad esempio, nell'intervallo $[0, 4294967295]$ con 32 bit)
- Esempio (con 8 bit):
  00001001 rappresenta il numero 9 ($2^3 + 2^0 = 8 + 1 = 9$)

Esempi: operazioni tra interi positivi

```
 95 +     01011111       64+16+8+4+2+1
 54       00110110       32+16+4+2
-------------------------------------
149       10010101       128+16+4+1

149 -     10010101       128+16+4+1
095       01011111       64+16+8+4+2+1
-------------------------------------
 54       00110110       32+16+4+2
```

N.B.: Se eccede il range, il bit di riporto si perde (overflow) $\Longrightarrow$ aritmetica modulo 2^N

```
223 +     11011111       128+64+16+8+4+2+1
 54       00110110       32+16+4+2
-------------------------------------
 21       00010101       16+4+1   (223+54-256=21)
```

Esempi: operazioni tra interi positivi

```
 95 +    01011111      64+16+8+4+2+1
 54      00110110      32+16+4+2
-------------------------------------
149      10010101      128+16+4+1

149 -    10010101      128+16+4+1
095      01011111      64+16+8+4+2+1
-------------------------------------
 54      00110110      32+16+4+2
```

N.B.: Se eccede il range, il bit di riporto si perde (overflow) $\Longrightarrow$ aritmetica modulo 2^N

```
223 +    11011111      128+64+16+8+4+2+1
 54      00110110      32+16+4+2
-------------------------------------
 21      00010101      16+4+1   (223+54-256=21)
```

Interi con segno

- I numeri interi con segno sono rappresentati tramite diverse codifiche
- Le codifiche più usate sono:
  - codifica segno-valore
  - codifica complemento a 2

Codifica Segno-valore

- Il primo bit rappresenta il segno, gli altri il valore
  - Esempio (8 bit): 10001001 rappresenta -9
- Comporta una doppia rappresentazione dello zero: 00000000 e 10000000
- I numeri rappresentati appartengono all'intervallo $[-2^{n-1}+1, 2^{n-1}-1]$ (ad esempio $[-2147483647, 2147483647]$ con 32 bit)
- poco usato in pratica

Nota:

Occorre usare due diversi algoritmi per la somma a seconda che il segno degli addendi sia concorde o discorde

Esempi: operazioni in segno-valore

Se il segno è diverso, si stabilisce il maggiore dei due in valore assoluto, e si calcolano somme o differenze.

```
         v
149 +    010010101       128+16+4+1
-95      101011111       64+16+8+4+2+1
-----------------------------------------
 54      000110110       32+16+4+2
         ^
```

Codifica Complemento a 2

- di gran lunga la più usata
- Un numero negativo è ottenuto calcolando il suo complemento (si invertono zeri e uni) e poi aggiungendo 1
- I numeri rappresentati appartengono all'intervallo $[-2^{n-1}, 2^{n-1} - 1]$ (ad es. $[-2147483648, 2147483647]$ con 32 bit)
- Rappresentazione ciclica: $-X$ in complemento a 2 si scrive come si scriverebbe $2^n - X$ nella rappresentazione senza segno
- comporta un'unica rappresentazione dello zero: 00000000

Esempio:

- 11110111 in complemento a 2 rappresenta -9:

```
   9       compl.      +1
00001001 11110110 11110111
```

- 11110111 in rappresentazione senza segno rappresenta 247, cioè $2^8 - 9$

Interi senza segno vs. complemento a 2

Codifica	senza segno		complemento a 2	
00000000		0		0
00000001		1		1
⋮		⋮		⋮
00001001	$8+1=$	9	$8+1=$	9
⋮		⋮		⋮
01111111	$64+\ldots+2+1=$	127	$64+\ldots+2+1=$	127
10000000		128	$128-256=$	-128
10000001	$128+1=$	129	$128+1-256=$	-127
⋮		⋮		⋮
10001001	$128+8+1=$	137	$128+8+1-256=$	-119
⋮		⋮		...
11111111	$128+\ldots+1=$	255	$128+\ldots+1-256=$	-1

Esempi: Operazioni in Complemento a 2

```
   -9 +    11110111
    9      00001001
=   0      00000000

   -9 +    11110111
    8      00001000
=  -1      11111111    => 11111110 => 00000001

   -9 +    11110111
   10      00001010
=   1      00000001

   -9 +    11110111
   -5      11111011
=  -14     11110010   => 11110001 => 00001110
```

Numeri Reali

Rappresentazione in virgola mobile (standard IEEE 754):

- $s \cdot m \cdot 2^{e-o}$, s è il segno ($\{-1, +1\}$), m è la mantissa, e è l'esponente e o è l'offset

Formato	*segno*	*esponente*	*mantissa*	*range*	*precisione*
32 *bit* :	1 *bit*	8 *bit*	23 *bit*	$\approx [10^{-38}, 10^{38}]$	$\approx$ 6 *cifre dec.*
64 *bit* :	1 *bit*	11 *bit*	52 *bit*	$\approx [10^{-308}, 10^{308}]$	$\approx$ 16 *cifre dec.*
128 *bit* :	1 *bit*	15 *bit*	112 *bit*	$\approx [10^{-4932}, 10^{4932}]$	$\approx$ 34 *cifre dec.*

(nota pratica: $2^{10} = 1024 \approx 1000 \implies 2^{10 \cdot N} \approx 10^{3 \cdot N}$)

- per la mantissa uso di codifica binaria decimale
  Es: "`.100110011001...`" significa "$1 \cdot \frac{1}{2} + 0 \cdot \frac{1}{4} + 0 \cdot \frac{1}{8} + 1 \cdot \frac{1}{16} + ...$"
  nota: (rappresentazione normale) primo bit assunto essere 1:
  Es: "000100001..." significa ".1000100001..."
- La codifica è dipendente dal numero di bit e dalla particolare rappresentazione binaria
- Richiede algoritmi ad-hoc per eseguire le operazioni

Outline

I Tipi

- Oggetti dello stesso tipo
  - utilizzano lo stesso spazio in memoria e la stessa codifica
  - sono soggetti alle stesse operazioni, con lo stesso significato
- Vantaggi sull'uso dei tipi:
  - correttezza semantica
  - efficiente allocazione della memoria dovuta alla conoscenza dello spazio richiesto in fase di compilazione

Tipi Fondamentali e Derivati in C++

Nel C++ i tipi sono distinti in:

- i tipi fondamentali
  - che servono a rappresentare informazioni semplici
  - Esempio: i numeri interi o i caratteri (int, char, ...)
- i tipi derivati
  - permettono di costruire strutture dati complesse
  - si costruiscono a partire dai tipi fondamentali, mediante opportuni costruttori (puntatori, array, ...)

I Tipi Fondamentali in C++

- i tipi interi: `int`, `short`, `long`, `long long`
- i tipi Booleani: `bool`
- i tipi enumerativi: `enum`
- il tipo carattere: `char`
- i tipi reali: `float`, `double`, `long double`

I primi quattro sono detti tipi discreti

I tipi interi (con segno)

- i tipi interi (con segno) sono costituiti da numeri interi compresi tra due valori estremi, a seconda dell'implementazione
  - tipicamente codificati in complemento a 2 con N bit (N = 16, 32, ...)
  - appartengono all'intervallo $[-2^{N-1}, 2^{N-1} - 1]$
- Quattro tipi, in ordine crescente di dimensione
  `short [int]`, `int`, `long [int]`, `long long [int]`
- Dimensioni dipendenti dall'implementazione (macchina, s.o., ...)
  - sizeof(short) $\leq$ sizeof(int) $\leq$ sizeof(long) $\leq$ sizeof(long long)
    (sizeof() restituisce la dimensione del tipo in byte)
  - `short` tipicamente non ha più di 16 bit: $[-32768, 32767]$
  - `long` tipicamente ha almeno 32 bit: $[-2147483648, 2147483647]$

Il valore di un'espressione intera non esce dal range $[-2^{N-1}, 2^{N-1} - 1]$ perché i valori "ciclano": il successore di 2147483647 è -2147483648

Esempio di short, int, long, long long:
{ `ESEMPI_BASE/shortlong.cc` }

I tipi interi (con segno)

- i tipi interi (con segno) sono costituiti da numeri interi compresi tra due valori estremi, a seconda dell'implementazione
  - tipicamente codificati in complemento a 2 con N bit (N = 16, 32, ...)
  - appartengono all'intervallo $[-2^{N-1}, 2^{N-1} - 1]$
- Quattro tipi, in ordine crescente di dimensione
  `short [int]`, `int`, `long [int]`, `long long [int]`
- Dimensioni dipendenti dall'implementazione (macchina, s.o., ...)
  - sizeof(short) $\leq$ sizeof(int) $\leq$ sizeof(long) $\leq$ sizeof(long long)
    (sizeof() restituisce la dimensione del tipo in byte)
  - `short` tipicamente non ha più di 16 bit: $[-32768, 32767]$
  - `long` tipicamente ha almeno 32 bit: $[-2147483648, 2147483647]$

Il valore di un'espressione intera non esce dal range $[-2^{N-1}, 2^{N-1} - 1]$ perché i valori "ciclano": il successore di 2147483647 è -2147483648

Esempio di short, int, long, long long:
{ `ESEMPI_BASE/shortlong.cc` }

I tipi interi (con segno)

- i tipi interi (con segno) sono costituiti da numeri interi compresi tra due valori estremi, a seconda dell'implementazione
  - tipicamente codificati in complemento a 2 con N bit (N = 16, 32, ...)
  - appartengono all'intervallo $[-2^{N-1}, 2^{N-1} - 1]$
- Quattro tipi, in ordine crescente di dimensione
  `short [int]`, `int`, `long [int]`, `long long [int]`
- Dimensioni dipendenti dall'implementazione (macchina, s.o., ...)
  - sizeof(short) $\leq$ sizeof(int) $\leq$ sizeof(long) $\leq$ sizeof(long long)
    (sizeof() restituisce la dimensione del tipo in byte)
  - `short` tipicamente non ha più di 16 bit: $[-32768, 32767]$
  - `long` tipicamente ha almeno 32 bit: $[-2147483648, 2147483647]$

Il valore di un'espressione intera non esce dal range $[-2^{N-1}, 2^{N-1} - 1]$ perché i valori "ciclano": il successore di 2147483647 è -2147483648

Esempio di short, int, long, long long:
{ `ESEMPI_BASE/shortlong.cc` }

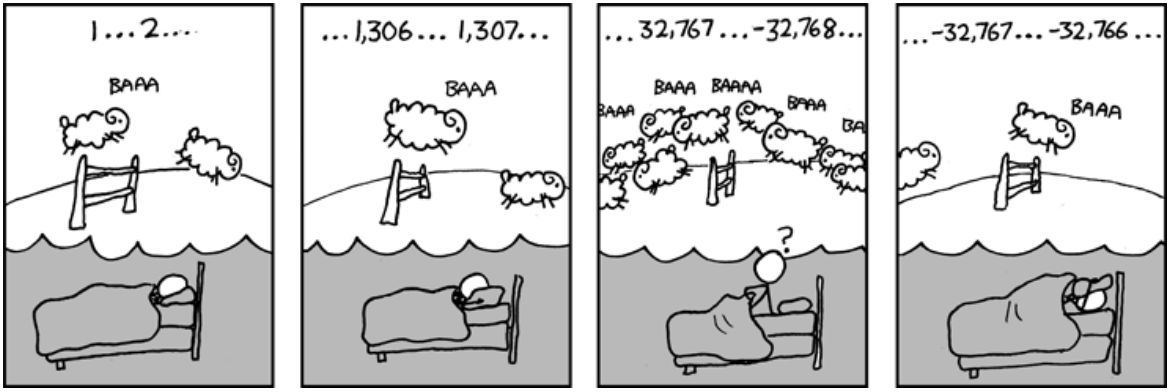
I tipi interi (con segno)

- i tipi interi (con segno) sono costituiti da numeri interi compresi tra due valori estremi, a seconda dell'implementazione
  - tipicamente codificati in complemento a 2 con N bit (N = 16, 32, ...)
  - appartengono all'intervallo $[-2^{N-1}, 2^{N-1} - 1]$
- Quattro tipi, in ordine crescente di dimensione
  `short [int]`, `int`, `long [int]`, `long long [int]`
- Dimensioni dipendenti dall'implementazione (macchina, s.o., ...)
  - sizeof(short) $\leq$ sizeof(int) $\leq$ sizeof(long) $\leq$ sizeof(long long)
    (sizeof() restituisce la dimensione del tipo in byte)
  - `short` tipicamente non ha più di 16 bit: $[-32768, 32767]$
  - `long` tipicamente ha almeno 32 bit: $[-2147483648, 2147483647]$

Il valore di un'espressione intera non esce dal range $[-2^{N-1}, 2^{N-1} - 1]$ perché i valori "ciclano": il successore di 2147483647 è -2147483648

Esempio di short, int, long, long long:
{ `ESEMPI_BASE/shortlong.cc` }

Sogni & incubi (a 16 bit) di un programmatore C++



I tipi interi (senza segno)

- Il tipo `unsigned...` rappresenta numeri interi non negativi di varie dimensioni
- codifica interi senza segno a N bit (N=16,32, ...), range $[0, 2^N - 1]$
- tipicamente usati molto poco, come rappresentazioni di sequenze di bit (applicazioni in elettronica)

```
unsigned int x=1232;
unsigned short int x=567;
unsigned long int x=878678687;
```

Il valore di un'espressione unsigned non esce mai dal range $[0, 2^N - 1]$ perché, quando il valore aritmetico X esce da tale range, il valore restituito è X modulo 2^N.

Esempio di unsigned:
{ `ESEMPI_BASE/unsigned.cc` }

Operatori aritmetici sugli interi

operatore binario (infisso)	significato
`+`	addizione
`-`	sottrazione
`*`	moltiplicazione
`/`	divisione intera
`%`	resto della div. intera
operatore unario (prefisso)	**significato**
`-`	inversione di segno

Nota:

la divisione è la divisione intera: `5/2` è `2`, non `2.5`!

Esempio sugli operatori aritmetici sugli interi:

{ `ESEMPI_BASE/operatori_aritmetici.cc` }

Operatori bit-a-bit (su interi senza segno)

operatore	esempio	significato
`>>`	`x>>n`	shift a destra di n posizioni
`<<`	`x<<n`	shift a sinistra di n posizioni
`&`	`x&y`	AND bit a bit tra x e y
`|`	`x|y`	OR bit a bit tra x e y
`^`	`x^y`	XOR bit a bit tra x e y
`~`	`~x`	NOT, complemento bit a bit

Nota:

- `~`: restituisce un intero signed anche se l'input è unsigned

Operatori bit-a-bit : esempio

Siano x e y rappresentati su 16 bit

```
x:     0000000000001100   (12)
y:     0000000000001010   (10)
x|y:   0000000000001110   (14)
x&y:   0000000000001000   ( 8)
x^y:   0000000000000110   ( 6)
~x:    1111111111110011   (65523 oppure -13 )
x>>2:  0000000000000011   ( 3)
x<<2:  0000000000110000   (48)
```

Esempio sugli operatori bit-a-bit su interi senza segno:
{ ESEMPI_BASE/operatori_bitwise.cc }

Operatore di Assegnazione

- Sintassi dell'operatore di assegnazione: exp1 = exp2
  - exp1 deve essere un'espressione dotata di indirizzo
  - exp1 e exp2 devono essere di tipo compatibile
- il valore di *exp*2 viene valutato e poi assegnato a *exp*1

  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 }

Operatore di Assegnazione

- Sintassi dell'operatore di assegnazione: exp1 = exp2
  - exp1 deve essere un'espressione dotata di indirizzo
  - exp1 e exp2 devono essere di tipo compatibile
- il valore di *exp*2 viene valutato e poi assegnato a *exp*1
  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 }

Operatore di Assegnazione

- Sintassi dell'operatore di assegnazione: exp1 = exp2
  - exp1 deve essere un'espressione dotata di indirizzo
  - exp1 e exp2 devono essere di tipo compatibile
- il valore di *exp*2 viene valutato e poi assegnato a *exp*1

  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
(utilizza in modo ottimale funzionalità delle 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 dell'incremento
- `++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 l'incremento

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 e' 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" come gli operatori di incremento. Es:

```
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`

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 e' 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" come gli operatori di incremento. Es:

```
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`

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 e' 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" come gli operatori di incremento. Es:

```
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

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 e' 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" come gli operatori di incremento. Es:

```
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` I:

- 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 `bool` 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 `bool` 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 `bool` 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 `bool` 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 `bool` 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 `bool` 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 sottinsieme dei razionali
- anche i tipi reali hanno un range finito!

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 sottinsieme dei razionali
- anche i tipi reali hanno un range finito!

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 sottinsieme dei razionali
- anche i tipi reali hanno un range finito!

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 sottinsieme dei razionali
- anche i tipi reali hanno un range finito!

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 sottinsieme 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` }

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` }

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
```

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 con o senza parentesi
- 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/`

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/`

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/`