Fondamenti di Informatica
Modulo B
A.A. 2004/2005

- Docente: Prof. Alessandro Sperduti
- E-mail: sperduti@math.unipd.it
- Tel: 049-8275982
- Ricevimento: giovedì dalle 15 alle 17
  (Dip. Matematica Pura ed Applicata, via Belzoni 7)

- 12 ore di lezione [11:30-13:00]
  - Gennaio: 12, 13, 19, 20, 26, 27
- 8 ore di laboratorio (Paolotti, Aula B)
  - Gennaio: 12 [15:00-17:00], 19 [14:00-17:00], 26 [14:00-17:00]
- Pagina del corso:
  - http://www.math.unipd.it/~sperduti/fondamentiB.html

1

Sommario degli argomenti

2

Concetti di Base

Linguaggi di Programmazione

- Linguaggi per impartire istruzioni al processore
- Programma
  - sequenza di istruzioni
  - normalmente pensato per risolvere un problema di calcolo
  - al programma vengono forniti dei dati
  - il programma calcola eseguendo le istruzioni
  - il programma restituisce i risultati

3

Linguaggi di programmazione

- Linguaggi ad alto livello (rispetto all'Assembler)
- Linguaggi imperativi: rispecchiano le operazioni reali nell'architettura di Von Neumann
  - Scrivere un valore in una cella di memoria o in un registro
- Es.: Fortran (Formula Translator), 1954
- Algol, Cobol, APL, LISP ('60), Pascal, Prolog ('71), C ('74), Ada ('79), C++ ('85), Java ('94)

4

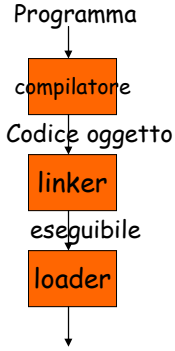
Elementi dei linguaggi di programmazione

- Alfabeto: alfabeto inglese, cifre decimali (0,1,...,9), punteggiatura (:), simboli di operazioni (+,-,*,...)
- Parole chiave: es. IF, THEN, WHILE, ...
- Operatori: aritmetici, logici (and, not, or, ...), relazionali (<, >, =, ...)
- Tipi di dati: interi, reali, caratteri, ...
- Sintassi: regole per scrivere un programma
- Semantica: significato dell'esecuzione di un programma

5

Da linguaggio X a linguaggio macchina

- Se X= Assembler ➔ assemblatore
- Se X piu' ad alto livello ➔ compilatore o interprete
- Fasi di un compilatore:
  - Analisi lessicale: da caratteri a nomi, operatori, parole chiave, ... (tokens)
  - Analisi sintattica/semantica: da tokens a costrutti (if then else, while, ...)
- Generazione codice oggetto in linguaggio macchina
- Linker/loader: collegamento tra vari programmi e scelta degli indirizzi di RAM



Programma in ling. macchina in RAM

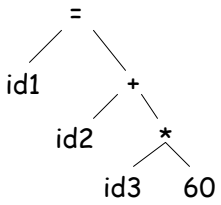
6

Esempio

- p = p + r * 60 ← sequenza di caratteri
- Analisi lessicale:

id1 = id2 + id3 * 60 ← sequenza di tokens

- Analisi sintattica/semantica
- Generazione codice intermedio: →



  - temp1 = 60
  - temp2 = id3*temp1
  - temp3 = id2 + temp2
  - id1 = temp3
- Ottimizzazione codice:
  - temp1 = id3 * 60
  - id1 = id2 + temp1
- Generazione codice oggetto:
  - LOAD R2 id3
  - LOAD R3 60
  - MULT R2 R3
  - LOAD R1 id2
  - ADD R1 R2
  - STORE R1 id1

7

Struttura di un programma C++

```
#include <iostream.h>
<altre eventuali direttive>

void main ()
{
   <dichiarazioni>

   <operazioni>

}
<funzione1>
<funzione2>
 …
```

8

Un primo esempio di programma

```
// il mio primo programma in C++

   #include <iostream.h>

   int main ()
   {
     cout << "Salve gente!";
     return 0;
   }
```

```
Salve gente!
```

Nella parte sinistra è riportato il codice del nostro primo programma che noi chiameremo, ad esempio, salve.cpp.

La parte destra mostra quello che si ottiene eseguendolo.

Comando di compilazione in Linux: **`g++ -o salve salve.cpp`**

- Richiama il compilatore g++ per compilare il programma scritto nel file salve.cpp
- Il risultato viene scritto nel file eseguibile **`salve`**

9

Un primo esempio di programma

```
// il mio primo programma in C++

   #include <iostream.h>

   int main ()
   {
     cout << "Salve gente!";
     return 0;
   }
```

Riga di commento. Tutte le righe che iniziano con due sbarrette (**//**) vengono considerate commento e non hanno alcun effetto sul comportamento del programma.

Direttiva. Le frasi che iniziano con il simbolo di cancelletto (#) sono direttive per il preprocessore del compilatore.
In questo caso la direttiva richiede di includere il file della libreria standard **`iostream.h`** che contiene le dichiarazioni delle operazioni basilari di input-output definite nella libreria standard del C++

La funzione **main** è il punto da cui inizia l'esecuzione di un qualsiasi programma C++. **main** è seguito da una coppia di parentesi **( )** perché esso è una funzione. In C++ tutte le funzioni sono seguite da una coppia di parentesi **( )** che possono contenere degli argomenti. La dichiarazione (l'intestazione) della funzione è seguita dal contenuto (il corpo) della funzione racchiuso tra parentesi graffe (**{ }**).

10

Un primo esempio di programma

```
// il mio primo programma in C++

   #include <iostream.h>

   int main ()
   {
     cout << "Salve gente!";
     return 0;
   }
```

cout è il flusso standard di output del C++ (di solito indirizzato allo schermo), e l'effetto della istruzione << è quello di inserire una sequenza di caratteri (nel nostro caso "Salve gente!") in tale flusso di output.

La dichiarazione di **cout** si trova nel file **iostream.h**, ed è per questo che è stato incluso tale file.

La frase finisce con un punto e virgola ";", che indica la fine dell'istruzione e deve essere messo alla fine di ogni istruzione (uno degli errori più comuni è appunto dimenticare il punto e virgola alla fine di una istruzione.

L'istruzione **return** fa terminare la funzione **main()** e ritorna quello che è indicato di seguito, nel nostro caso **0**.
Poiché il valore ritornato è un intero bisogna dichiarare **main** come una funzione che ritorna un intero (**int main ()**).
Molti compilatori inseriscono automaticamente una istruzione **return** alla fine di una funzione.

11

Un primo esempio di programma

Il corpo del programma è stato diviso in più righe per renderlo più leggibile, ma si può scrivere lo stesso programma in un'unica riga:

```
int main () { cout << "Salve gente!"; return 0;}
```

- In C++ la separazione tra istruzioni è indicata dal punto e virgola che termina ciascuna di esse. La suddivisione del programma in più righe serve a rendere il programma più leggibile alle persone che lo leggono, per il compilatore la cosa non fa' alcuna differenza.
- Questa regola non vale per le direttive per il preprocessore (quelle che iniziano con **#**) in quanto esse sono istruzioni per il preprocessore che verranno lette ed interpretate dal preprocessore e che spariranno dal codice che verrà compilato dal compilatore.
- Esse devono comparire ciascuna in una propria riga e non richiedono il punto e virgola alla fine.

12

Un secondo esempio di programma

Ecco un programma con qualche altra istruzione:

```
// il mio secondo programma in C++

   #include <iostream.h>

   int main ()
   {
     cout << "Salve gente!";
     cout << “Sono un programma C++”;
     return 0;
   }
```

```
Salve gente! Sono un programma C++
```

Il cui corpo può essere anche scritto in una sola riga:

```
int main () { cout << "Salve gente!"; cout << “Sono un programma C++”; return 0;}
```

13

Commenti

I commenti sono parti del testo che vengono ignorate dal compilatore.
In C++ ci sono due modi di inserire commenti:

```
// riga di commento
/* blocco di commento */
```

Il primo considera commento tutto ciò che segue la coppia di sbarre (**//**) fino alla fine della riga. Il secondo considera commento tutto ciò che è compreso tra la coppia di caratteri **/*** e la prima occorrenza successiva della coppia di caratteri ***/**, **eventualmente anche più righe di testo**.

```
/* il mio secondo programma in C++
   con più commenti */

   #include <iostream.h>

   int main ()
   {
     cout << "Salve gente!"; // dice Salve gente!
     cout << “Sono un programma C++”;  // dice Sono un programma C++
     return 0;
   }
```

14

Variabili

- Variabile: nome associato ad una cella di memoria
- Può assumere un valore (quello contenuto nella cella)
- Istruzione di assegnamento per dare un valore ad una variabile (es: `a = 5`)
- Valore ottenuto calcolando un'espressione (es: `a + 1`)

Es.

```
a = 5;
b = 2;
a = a + 1;
risultato = a - b;
```

Ogni variabile necessita di un identificatore (un nome) che la distingue da ogni altra variabile.
Ad esempio nel codice precedente gli identificatori sono `a` , `b` e `risultato`, ma avremmo potuto chiamare le variabili con un qualsiasi altro nome inventato da noi, purchè esso sia un identificatore valido.

15

Identificatori

- Un identificatore valido è una sequenza di una o più lettere, cifre o simboli di sottolineatura (_).
- La lunghezza di un identificatore non è limitata, ma alcuni compilatori considerano significativi soltanto i primi 32 caratteri .
- Un identificatore non deve contenere spazi o altri caratteri. Sono permessi soltanto lettere, cifre e simboli di sottolineatura. Inoltre, gli identificatori di variabile devono sempre iniziare con una lettera.
- In nessun caso un identificatore può iniziare con una cifra.
- Un'altra regola da tener presente quando si scelgono gli identificatori è che essi non devono essere uguali ad una delle **parole chiave del linguaggio** nè ad una di quelle specifiche del compilatore usato.

16

Tipi di Dato

- I valori di una variabile (un carattere, un piccolo numero, un grande numero,…) possono essere di vario tipo e quindi possono occupare una quantità di memoria diversa nel calcolatore
- La memoria di un calcolatore è suddivisa in byte. Il byte è la più piccola quantità di memoria che possiamo gestire, esso può contenere una relativamente piccola quantità di dati: di norma un singolo carattere o un intero fra 0 e 255.
- Il calcolatore può elaborare anche tipi di dato più complessi che occupano più di un byte.
- Vediamo di seguito un elenco dei tipi fondamentali del C++ con l'indicazione della quantità di memoria necessaria e del rango di valori che si possono rappresentare:

17

Tipi di Dato

Nome	Byte*	Descrizione	Rango*
char	1	carattere o intero di 8 bit.	**signed:** -128 ... 127 **unsigned:** 0 ... 255
short	2	intero di 16 bit.	**signed:** -32768 ... 32767 **unsigned:** 0 ... 65535
long	4	intero di 32 bit.	**signed:**-2147483648 ... 2147483647 **unsigned:** 0 ... 4294967295
int	*	Intero. La sua lunghezza dipende dalla lunghezza del tipo **word** usato dal sistema operativo. Ad esempio, in MSDOS è di 16 bit mentre in sistemi a 32 bit (quali Windows 9x/2000/NT) è di 32 bit (4 bytes).	Vedi **short**, **long**
float	4	numero in virgola mobile.	3.4e + / - 38 (7 cifre decimali)
double	8	numero in virgola mobile in doppia precisione.	1.7e + / - 308 (15 cifre decimali)
long double	10	numero in virgola mobile in doppia precisione estesa.	1.2e + / - 4932 (19 cifre decimali)
bool	1	Valori Booleani. Può assumere uno dei due valori: true o false. NOTA: è un tipo aggiunto recentemente allo standard ANSI-C++. Non tutti i compilatori lo accettano.	**true** or **false**
wchar_t	2	Carattere esteso. Viene usato per rappresentare tutti i caratteri internazionali. NOTA: è un tipo introdotto recentemente nello standard ANSI-C++. Non tutti i compilatori lo accettano.	caratteri estesi

* I valori nelle colonne Byte e Range possono variare a seconda del sistema.

18

Dichiarazioni delle Variabili

- Prima di usare una variabile occorre dichiararla specificando a quale tipo di dato essa appartenga.
- La sintassi di una dichiarazione di variabile prevede prima il nome del tipo di dato (quale **int**, **short**, **float**, ...) seguito dall'identificatore scelto per denotare tale variabile. Ad esempio:

```
int a;
float
numero;
```

- Una volta dichiarate, le variabili **a** e **numero** possono essere usate nel programma all'interno del loro campo di validità (lo scope).
- Se vogliamo dichiarare più di una variabile dello stesso tipo possiamo farlo in una stessa riga indicando una sola volta il tipo e separando gli identificatori con la virgola. Ad esempio:

```
int a, b, c;
```

19

Dichiarazioni delle Variabili

- I tipi di dato interi (**char**, **short** , **long** e **int**) possono essere signed o unsigned a seconda del rango di numeri che si vuole considerare.
- Per specificare un intero possiamo usare una delle parole chiave **signed** o **unsigned** prima del nome del tipo. Ad esempio:

```
unsigned short NumeroDiFigli;
signed int IlMioSaldoBancario;
```

- Se non si specifica né **signed** né **unsigned** viene assunto **signed**
- Possiamo usare **signed** e **unsigned** come nomi di tipo con lo stesso significato di **signed int** e **unsigned int** rispettivamente. Le seguenti due dichiarazioni sono equivalenti:

```
unsigned AnnoDiNascita;
unsigned int AnnoDiNascita;
```

20

Un altro esempio di programma

```
// operazioni con le variabili
   #include <iostream.h>
   int main ()
   {
     // dichiarazione delle variabili:
     int a, b;
     int risultato;
     // elaborazione:
     a = 5;
     b = 2;
     a = a + 1;
     risultato = a - b;
     // stampa del risultato:
     cout << risultato;
     // terminazione del programma:
     return 0;
   }
```

```
4
```

21

Inizializzazione delle Variabili

- Quando dichiariamo una variabile il suo valore è indeterminato
  (i bit della zona di memoria riservata alla variabile hanno il valore che era stato loro assegnato da qualche precedente programma).
- Se nel dichiarare una variabile vogliamo anche assegnarle un valore iniziale, basta aggiungere alla dichiarazione un simbolo di uguale seguito dal valore desiderato:

```
tipo identificatore = valore_iniziale ;
```
← Stile C

```
tipo identificatore(valore_iniziale);
```
← Stile C++

- Ad esempio se vogliamo dichiarare una variabile a di tipo **int** con valore iniziale 0 possiamo scrivere:

```
int a = 0;
```
← Stile C

```
int a(0);
```
← Stile C++

In C++ si può utilizzare sia lo stile C che lo stile C++

22

Scope delle Variabili

```
#include <iostream.h>

int Intero;
char unCarattere;
char stringa[20];
unsigned int NumeroFigli;

main ()
{
float unNumero, altroNumero;
unsigned short Eta;

cout << "scrivi la tua eta ";
cin >> Eta;
...
}
```

variabili globali

variabili locali

istruzioni

Le **variabili globali** si possono usare in tutto il programma dal punto in cui sono dichiarate fino alla fine.

Lo scopo delle **variabili locali** è invece limitato al blocco in cui sono dichiarate. Se sono dichiarate all'inizio di una funzione (come in **main**) il loro scopo è l'intero corpo della funzione. Se nell'esempio ci fosse un'altra funzione diversa da **main()**, le variabili locali dichiarate in **main** non sarebbero visibili all'interno di tale funzione e viceversa.

In C++ lo scopo di una variabile locale è limitato alla parte del blocco in cui esse sono dichiarate che segue la dichiarazione stessa (un blocco è un gruppo di istruzioni racchiuse tra parentesi graffe {}.

23

Costanti

- Una costante è una qualsiasi espressione che ha un valore prefissato.
- Esse si possono suddividere in Numeri Interi, Numeri in Virgola Mobile, Caratteri e Stringhe.
- Per scrivere una costante intera non occorre usare le virgolette (") o qualche altro carattere speciale:

```
75    // decimale
0113  // 75 in ottale (occorre premettere il carattere 0)
0x4b  // 75 in esadecimale (occorre premettere i due caratteri 0x)
```

- Numeri in virgola mobile (numeri con una parte frazionaria e/o un fattore esponenziale):

```
3.14159     // 3.14159
6.02e23     // 6.02 x 10^23
1.6e-19     // 1.6 x 10^-19
3.0         // 3.0
```

24

Costanti

- Caratteri e Stringhe:

```
'z'
'p'
"Salve gente"
"Come state?"
```

caratteri

stringhe

- I letterali che rappresentano singoli caratteri sono racchiusi tra due caratteri apice (') mentre i letterali che rappresentano stringhe sono racchiusi tra due caratteri doppio apice (").
- Questo è necessario per poter distinguere valori di tipo stringa da valori di tipo carattere (che sono considerati diversi).
- Inoltre, l'uso degli apici (') e (") evita di confondere un letterale carattere o stringa da un identificatore o una parola chiave:
  x denota la variabile x, mentre `'x'` denota la costante di tipo carattere 'x'

25

Costanti

- Per rappresentare con un letterale (o all'interno di un letterale stringa) alcuni caratteri speciale si usano notazioni particolari (i codici di escape).
- Ecco una lista di tali codici di escape (ognuno di essi inizia con il carattere barra rovesciata (\):

\n	a capo riga	**\r**	ritorno carrello	**\t**	tabulazione
\v	tabulazione verticale	**\b**	backspace	**\f**	nuova pagina
\a	allerta (beep)	**\'**	apice singolo (')	**\"**	doppio apice (")
\?	punto interrogativo (?)	****	barra rovesciata (\ )		

Esempio:

```
'\n'
'\t'
"Sinistra \t Destra"
"uno\ndue\ntre"
```

26

Costanti Definite (#define)

- Possiamo usare la direttiva **#define** del preprocessore per dare un nome ad una costante nel seguente modo:

```
#define identificatore_di_costante
```

- Ovunque useremo l'identificatore, il preprocessore provvederà a sostituirlo con la costante.

```
#define PI 3.14159265
#define NEWLINE '\n'
#define WIDTH 100
```

Esempio di uso:

```
circonferenza = 2 * PI * r;
cout << NEWLINE;
```

- **#define** non è una istruzione C++ ma una direttiva per il preprocessore: deve quindi essere scritta in una sua propria riga e non deve essere aggiunto il carattere ; alla fine.
- Una volta che abbiamo associato un identificatore ad una costante possiamo usare tale identificatore in ogni punto seguente del programma come se esso fosse una costante.

27

Costanti Dichiarate (const)

- Usando il prefisso **const** si possono dichiarare delle costanti appartenenti ad un determinato tipo esattamente allo stesso modo in cui si dichiarano le variabili:

```
const int larghezza = 100;
const char tab = '\t';
const int cap =  12440;
```

- In realtà le costanti dichiarate sono semplicemente delle variabili il cui valore non può più essere modificato
- Siccome una volta create non è più possibile cambiarne il valore, esse devono essere sempre inizializzate con un valore al momento della loro creazione.

28

Operatori

- Il C++ fornisce degli operatori per la manipolazione di variabili e costanti, che nel nostro linguaggio sono un insieme di parole chiave e simboli speciali.
- Gli operatori che vedremo sono:
  - Assegnamento (=)
  - Operatori aritmetici (+, -, *, /, %)
  - Operatori di assegnazione composti (+=, -=, *=, /=, %=, >>=, <<=, &=, ^=, |=)
  - Incremento e decremento
  - Operatori relazionali (==, !=, >, <, >=, <=)
  - Operatori logici (!, &&, ||)
  - Operatore condizionale (?)
  - Operatori bit a bit (&, |, ^, ~, <<, >>)
  - Operatori espliciti di conversione di tipo (casting)
  - Operatore sizeof()

29

Assegnamento

- L'operatore di assegnamento serve per assegnare un valore ad una variabile `( a = 5;)`
- La parte alla sinistra dell'operatore = è nota come lvalue (left value) e la parte destra rvalue (right value)
- lvalue deve essere sempre una variabile mentre la parte destra può essere una costante, una variabile, il risultato di una operazione o una qualsiasi combinazione di essi
- L'operatore di assegnazione opera sempre e solo da destra verso sinistra
- `( a = b; )` assegna alla variabile **a** (lvalue) il valore della variabile **b** (rvalue) indipendentemente dal valore che era precedentemente memorizzato nella variabile **a**
- Possiamo pensare l'lvalue di **a** come l'indirizzo della zona di memoria riservata per memorizzare il valore di **a** mentre l'rvalue di **b** lo dobbiamo pensare come il valore memorizzato nella zona di memoria riservata per memorizzare il valore di **b**
- Notiamo inoltre che noi stiamo soltanto assegnando ad **a** il valore di **b** e che una modifica successiva di **b** non cambia il nuovo valore di **a**

30

Assegnamento

- Esempio di codice dove i commenti mostrano l'evoluzione del contenuto delle variabili:

```
int a, b;     // a:?  b:?
a = 10;       // a:10 b:?
b = 4;        // a:10 b:4
a = b;        // a:4 b:4
b = 7;        // a:4 b:7
```

- Una caratteristica dell'operatore di assegnamento presente in C++ è che un assegnamento si può usare come rvalue (o parte di un rvalue) in un altro assegnamento. Ad esempio:

```
a = 2 + (b = 5);
```
è equivalente a
```
b = 5;
a = 2 + b;
```

- Quindi, anche la seguente espressione è valida in C++:

```
a = b = c = 5;
```

31

Operatori aritmetici (e di assegnamento composti)

- I cinque operatori aritmetici previsti dal linguaggio sono:
  + somma - differenza * moltiplicazione / divisione % modulo (o resto)
- Gli operatori di somma, differenza, moltiplicazione e divisione denotano le usuali quattro operazioni numeriche.
- L'operatore di modulo fornisce il resto della divisione di due numeri interi. Ad esempio, se scriviamo (`a = 11 % 3;`), viene assegnato alla variabile **a** il valore 2 in quanto 2 è proprio il resto che si ottiene dividendo 11 per 3.

Operatori di assegnamento composti:

- permettono di modificare il valore di una variabile con una sola operazione:

`valore += incremento;`	è equivalente a	`valore = valore + incremento;`
`a -= 5;`	è equivalente a	`a = a - 5;`
`a /= b;`	è equivalente a	`a = a / b;`
`prezzo *= numero + 1;`	è equivalente a	`prezzo = prezzo * (numero + 1);`

32

Incremento e decremento

- Gli operatori di incremento (++) e di decremento (--) aumentano o diminuiscono di 1 il valore di una variabile e sono equivalenti a += 1 e -= 1 rispettivamente. Quindi:

  `a++;  a += 1;  a = a + 1;` sono istruzioni equivalenti
- Questi operatori si possono usare sia come prefissi che come postfissi, ossia possono essere scritti prima della variabile (`++a`) o dopo di essa (`a++`).
- ATTENZIONE !
  - se si usa `++a` , il valore della variabile viene incrementato prima della valutazione dell'espressione e quindi l'espressione viene valutata usando il valore incrementato;
  - se si usa `a++` , il valore della variabile viene incrementato dopo la valutazione dell'espressione e quindi l'espressione viene valutata usando il valore non incrementato.
- Ecco alcuni esempi della differenza tra i due modi di usare l'operatore:

```
b = 3;
a = ++b;
// a è 4, b è 4
```

```
b = 3;
a = b++;
// a è 3, b è 4
```

33

Operatori Relazionali

- Per confrontare i valori di due espressioni si usano gli operatori relazionali.
- Il risultato di un operatore relazionale è di tipo bool e può assumere soltanto uno dei due valori booleani true o false, a seconda del risultato del confronto.
- Gli operatori relazionali in C++ sono i seguenti:

  == Uguale != Diverso > Maggiore < Minore >= Maggiore o uguale

  <= Minore o uguale
- Ecco alcuni esempi:

  `(7 == 5)` risultato `false`
  `(3 != 2)` risultato `true`
  `(5 < 5)` risultato `false`

  `(5 > 4)` risultato `true`
  `(6 >= 6)` risultato `true`

  Supponendo che `a=2`, `b=3` e `c=6`

  `(a == 5)` risultato `false`
  `(a*b >= c)` risultato `true` in quanto (2*3 >= 6)
  `(b+4 > a*c)` risultato `false` in quanto (3+4 > 2*6)
  `((b=2) == a)` risultato `true`

  ATTENZIONE ! L'operatore = (un solo uguale) è differente dal simbolo == (doppio uguale)

34

Operatori Relazionali

ATTENZIONE !

In molti compilatori precedenti la pubblicazione dello standard ANSI-C++, come pure in C, gli operatori relazionali

- non ritornano un valore bool (true o false)
- ma ritornano un valore int con
  - 0 che rappresenta "false" e
  - un valore diverso da 0 (in genere 1) che rappresenta "true"

35

Operatori Logici

- L'operatore ! è l'operatore logico di negazione NOT.
- Esso ha un unico operando (di tipo bool) posto alla sua destra e il suo risultato è l'opposto del valore dell'operando:
  - se l'operando è true esso ritorna false,
  - se l'operando è false esso ritorna true.

Ad esempio:

!(5 == 5) ritorna false in quanto l'espressione alla sua destra (5 == 5) è true.
!(6 <= 4) ritorna true in quanto (6 <= 4) è false.
!true ritorna false.
!false ritorna true.

- Gli operatori && e || sono gli operatori logici di congiunzione (AND) e disgiunzione (OR). Essi hanno due operandi (di tipo bool), uno alla sinistra ed uno alla destra

Ad esempio:

((5 == 5) && (3 > 6)) ritorna false (true && false).
((5 == 5) || (3 > 6)) ritorna true (true || false) .

36

Operatore Condizionale

- L'operatore condizionale (?) valuta una espressione booleana e ritorna un valore diverso a seconda che tale valore sia true oppure false.
- La sua forma è:

```
condizione ? risultato1 : risultato2
```

- se **condizione** è true
  - l'espressione ritorna **risultato1**,
  - altrimenti ritorna **risultato2**.

Ad esempio

7==5 ? 4 : 3	ritorna 3 perché 7 non è uguale a 5 .
7==5+2 ? 4 : 3	ritorna 4 perché 7 è uguale a 5+2 .
5>3 ? a : b	ritorna a, perché 5 è maggiore di 3 .
a>b ? a : b	ritorna il maggiore dei due, a o b .

37

Operatori bit a bit

- Gli operatori bit a bit operano in parallelo su tutti i bit degli operandi.
- Essi sono operatori di basso livello a cui corrispondono alcune operazioni assembler:

operatore	assembler	Descrizione
&	**AND**	AND bit a bit
|	**OR**	OR bit a bit.
^	**XOR**	OR esclusivo bit a bit.
~	**NOT**	NOT bit a bit (complemento a uno).
<<	**SHL**	Spostamento dei bit (shift) a sinistra
>>	**SHR**	Spostamento dei bit (shift) a destra

- Nella programmazione normale C++ essi non dovrebbero essere usati.

38

Operatori espliciti di conversione di tipo (casting)

- Gli operatori di conversione servono per convertire valori appartenenti ad un tipo in valori di un altro tipo.
- Vi sono diversi modi per fare questo in C++

Stile C

- Si fa precedere le espressioni che devono essere convertite dal nome del nuovo tipo racchiuso tra parentesi ():

```
int i;
float f = 3.14;
i = (int) f;
```

Converte il numero float 3.14 nel valore intero 3.
L'operatore di conversione è rappresentato da (int).

Stile C++

- far precedere l'espressione da convertire, racchiusa tra parentesi, dal nome del nuovo tipo

```
int i;
float f = 3.14;
i = int( f );
```

Usa la funzione "costruttore".

39

Operatore sizeof()

- L'operatore sizeof() ha un parametro che può essere sia il nome di un tipo che una espressione.
- Esso ritorna la memoria, in byte, necessaria a memorizzare un valore di tale tipo o il valore dell'espressione:

```
a = sizeof (char);
```

ritorna 1 in **a** perché un valore di tipo char occupa un byte.

```
a = sizeof (double);
```

ritorna 8 in **a** perché un valore di tipo double occupa 8 byte.

- Il valore ritornato da sizeof() è una costante.
- Tale valore è quindi determinato a tempo di compilazione (prima dell'esecuzione del programma).
- Viene utilizzato (soprattutto in C) per stabilire quanta memoria allocare.

40

Priorità degli operatori

- Quando scriviamo espressioni complesse con molti operatori e operandi possono sorgere dei dubbi sull'ordine in cui gli operatori vengono valutati. Ad esempio, con l'espressione:

```
a = 5 + 7 % 2;
```

può sorgere il dubbio tra le seguenti due interpretazioni:

```
a = 5 + (7 % 2); // con risultato 6, o
a = (5 + 7) % 2; // con risultato 0
```

- L'interpretazione corretta è la prima, quella con risultato 6.
- Vi è un ordine di priorità prestabilito tra tutti gli operatori (sia aritmetici che non aritmetici).
- La precedenza degli operatori in una espressione si può modificare o rendere evidente usando le parentesi (e) .

41

Priorità degli operatori

Priorità	Operatore	Descrizione	Associatività
1	`::`	scopo	Sinistra
2	`() [ ] -> . sizeof`		Sinistra
3	`++ --`	incremento/decremento	Destra
	`~`	Complemento a uno (bit a bit)	
	`!`	NOT	
	`& *`	Referenziazione e Dereferenziazione (puntatori)	
	`(tipo)`	Conversione di tipo	
	`+ -`	Operatori di segno unario	
4	`* / %`	Operatori aritmetici moltiplicativi	Sinistra
5	`+ -`	Operatori aritmetici additivi	Sinistra
6	`<< >>`	Spostamento dei bit	Sinistra
7	`< <= > >=`	Operatori relazionali	Sinistra
8	`== !=`	Operatori relazionali di uguaglianza	Sinistra
9	`& ^ |`	Operatori bit a bit	Sinistra
10	`&& ||`	Operatori logici	Sinistra
11	`?:`	Operatore condizionale	Destra
12	`= += -= *= /= %= >>= <<= &= ^= |=`	Assegnamento	Destra
13	`,`	Operatore virgola, separatore	Sinistra

42

Comunicazione da console

- La console è l'interfaccia base del calcolatore, essa è di solito composta da una tastiera (usata come unità di input standard) ed un video (usato come unità di output standard).
- Nalla libreria standard **iostream** del C++ le operazioni di input ed output di un programma vengono gestite da due flussi di dati (stream):
  - **cin** per l'input (normalmente assegnato alla tastiera)
  - **cout** per l'output (normalmente diretto al video)
- Sono definiti inoltre altri due flussi - **cerr** e **clog** - il cui scopo è quello di segnalare eventuali messaggi di errore. Tali flussi possono essere mandati anch'essi sul video oppure inviati in un file di log.
- Usando questi due flussi un programma può interagire con un utente mostrando messaggi sullo schermo e ricevendo l'input da parte dell'utente dalla tastiera.

43

Output (cout)

- Il flusso **cout** viene usato assieme all'operatore "sovraccaricato" << (una coppia di segni di "minore"):

```
cout << "Frase di output"; // stampa Frase di output sullo schermo
cout << 120;              // stampa il numero 120 sullo schermo
cout << x;                // stampa il valore di x sullo schermo
```

- L'operatore << è noto come operatore di inserimento in quanto esso inserisce il valore alla sua destra nel flusso di dati indicato alla sua sinistra.
- Nel primo esempio esso inserisce nel flusso di output standard **cout** la stringa costante Frase di output racchiusa tra doppi apici ("), questo perché essa è una stringa di caratteri.
- Quando vogliamo usare una stringa costante di caratteri dobbiamo racchiuderla tra doppi apici per distinguerla da un identificatore di variabile:

```
a cout << "Salve"; // stampa Salve sullo schermo
cout << Salve;     // stampa il valore della variabile Salve sullo schermo
```

Output (cout)

- L'operatore di inserzione (<<) può essere usato più di una volta nella stessa frase. Ad esempio:

```
cout << "Salve, " << "io sono " << "una frase C++";
```

L'utilità di poter usare più volte l'operatore di inserzione nella stessa frase si vede quando dobbiamo stampare una combinazione di costanti e variabili o anche semplicemente più di una variabile:

```
cout << "Salve, io ho " << eta << " anni e il mio CAP e' " << cap;
```

- Se assumiamo che la variabile **eta** abbia il valore 24 e che la variabile **cap** abbia valore 65064 l'output che si ottiene è:

Salve, io ho 24 anni e il mio CAP e' 65064

- Attenzione: **cout** non va a capo dopo aver stampato a meno che non glielo si dica esplicitamente. Pertanto, con le due seguenti frasi:

```
cout << "Questa e' una frase.";
cout << "Questa e' un'altra frase.";
```

- otteniamo:

Questa e' una frase.Questa e' un'altra frase.

anche se sono state scritte con due distinte chiamate a **cout**

45

Output (cout)

- Per andare a capo bisogna inserire esplicitamente nel flusso un carattere speciale di nuove-linea che, in C++, si indica con **\n** :

```
cout << "Prima frase.\n ";
cout << "Seconda frase.\nTerza frase.";
```

stampa:

Prima frase.
Seconda frase.
Terza frase.

- Per andare a capo possiamo anche usare il *manipolatore* **endl** . Ad esempio:

```
cout << "Prima frase." << endl;
cout << "Seconda frase." << endl;
```

stampa:

Prima frase.
Seconda frase.

46

Input (cin)

- In C++ l'input standard si effettua applicando l'operatore di estrazione (>>) al flusso **cin** . Tale operatore deve essere seguito dalla variabile in cui deve essere memorizzato il valore da leggere. Ad esempio:

```
int eta;
cin >> eta;
```

dichiara la variabile **eta** di tipo **int** e quindi aspetta un input da **cin** (tastiera) per poter memorizzare un valore intero in tale variabile.

- ***Attenzione:*** **cin** elabora l'input ricevuto da tastiera soltanto dopo che è stato premuto il tasto di invio ENTER.
- Quindi, anche se viene richiesto di leggere un singolo carattere, **cin** non elabora l'input fino a quando non viene premuto anche il tasto ENTER.

- Quando si usa l'estrattore (>>) su **cin** bisogna tener presente il tipo della variabile che si usa per memorizzare il valore letto.
- Se viene richiesto un intero deve essere introdotto un intero, se viene richiesto un carattere deve essere introdotto un carattere e se viene richiesta una stringa deve essere introdotta una stringa.

47

Input (cin)

```
// esempio di i/o

#include <iostream.h>

int main ()
 {
  int i;
  cout << "Dammi un intero: ";
  cin >> i;
  cout << "Il valore che mi hai dato e' " << i;
  cout << " e il suo doppio e' " << i*2 << ".\n";
  return 0;
 }
```

```
Dammi un intero: 702
Il valore che mi hai dato e' 702 e il suo doppio e' 1404.
```

48

Input (cin)

- Si può anche usare **cin** per richiedere più di un dato alla volta:

```
cin >> a >> b;
```

è equivalente a:

```
cin >> a;
cin >> b;
```

- In entrambi i casi l'utente deve fornire due valori dei tipi appropriati, uno per la variabile a ed uno per la variabile b.
- Tali valori possono essere separati da uno o più caratteri spazio o tabulazione o nuova-linea.

49