Università della Calabria
Corso di Laurea in Ingegneria Elettronica

Programmazione Orientata agli Oggetti

A.A. 2006/2007

Fondamenti di C++

Paolo Trunfio *

* DEIS, Università della Calabria – http://si.deis.unical.it/~trunfio

Input/Output di base

La libreria standard *iostream* fornisce le operazioni di base per l'input e per l'output.

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

int main() {
  int a, b;
  cout << "Primo numero: ";
  cin >> a;
  cout << "Secondo numero: ";
  cin >> b;
  int somma = a+b;
  cout << "La somma dei due numeri e' " << somma << endl;
}
```

Commenti

Sono disponibili due tipi di commento: commenti di una sola riga e commenti multiriga.

I commenti di una sola riga iniziano con la doppia barra (//) e terminano con la fine della riga.

I commenti multilinea iniziano con la sequenza /* e terminano con la sequenza */, e possono estendersi su più righe.

Esempi:

```
int a = 4;  // commento di una sola riga

int b = 8;
/* esempio di
   commento
   multiriga
*/
```


Tipi predefiniti

Tipo	Numero di bit *	Valori rappresentati *
bool	8	Booleani: false e true
char	8	Caratteri ASCII o interi: [-128, 127]
unsigned char	8	Caratteri ASCII o interi: [0, 255]
short	16	Interi: [-32768, 32767]
unsigned short	16	Interi: [0, 65535]
int	32	Interi: [-2147483648, 2147483647]
unsigned int	32	Interi: [0, 4294967295]
long	32	Interi: [-2147483648, 2147483647]
unsigned long	32	Interi: [0, 4294967295]
float	32	Reali (7 cifre significative): [-3.4e+38, +3.4e+38]
double	64	Reali (15 cifre signif.): [-1.8e+308, +1.8e+308]
long double	96	Reali (19 cifre signif.): [-1.2e+4932, +1.2e+4932]

*** Dipendente dal sistema: i dati riportati sono riferiti ad un sistema a 32 bit.**

Tipi signed

I tipi interi *char*, *short*, *int* e *long* possono essere esplicitamente dichiarati *signed* (con segno):

- signed char
- signed short
- signed int
- signed long

Il termine *signed* per *short*, *int* e *long* viene generalmente omesso, in quanto questi tipi sono sempre con segno, a meno che non si specifichi in modo esplicito che sono *unsigned*.

Il tipo *signed char* è invece utilizzato in alcune implementazioni nelle quali il tipo *char* è implementato come un *unsigned char*.

Nomi di tipo estesi

I tipi *short* e *long* possono essere considerati due varianti del tipo *int*.

In particolare, il termine *short* è un'abbreviazione di *short int*, mentre *long* è un'abbreviazione di *long int*.

Tenendo conto anche delle varianti *signed* e *unsigned*, valgono le seguenti equivalenze tra i nomi dei tipi:

- short = short int = signed short int
- int = signed int
- long = long int = signed long int
- unsigned short = unsigned short int
- unsigned long = unsigned long int

La versione abbreviata è preferibile, anche se i nomi di tipo estesi sono talvolta utilizzati.

L’operatore sizeof

La dimensione di un oggetto o di un tipo può essere ottenuta mediante l’operatore *sizeof*.

Il valore restituito da *sizeof* è un intero senza segno che esprime la dimensione come multiplo della dimensione di un *char*.

```
cout << sizeof(char);  // 1 (per definizione)

cout << sizeof(int);  // 4

cout << sizeof(unsigned int);  // 4

cout << sizeof(long double);  // 12

long a;

cout << sizeof(a);  // 4
```


Relazione tra le dimensioni dei tipi predefiniti

Sono definite le seguenti relazioni tra le dimensioni dei tipi predefiniti:

1 = sizeof(char) ≤ sizeof(short) ≤ sizeof(int) ≤ sizeof(long)

1 ≤ sizeof(bool) ≤ sizeof(long)

sizeof(float) ≤ sizeof(double) ≤ sizeof(long double)

Inoltre, la dimensione di un tipo con segno o senza segno è identica:

sizeof(T) = sizeof(signed T) = sizeof(unsigned T)

dove T può essere *char*, *short*, *int* o *long*.

Tipo bool

Una variabile di tipo *bool* può assumere i valori *false* e *true*, ed è utilizzata per esprimere il risultato di un’operazione logica.

```
bool a;
a = true;
if (a == true)  // if (a)
  cout << "vero";
```

Le variabili booleane possono essere convertite in variabili intere: *false* corrisponde a *0* e *true* corrisponde a *1*.
Allo stesso modo le variabili intere possono essere convertite in variabili booleane: *0* è convertito in *false* e qualsiasi intero diverso da *0* è convertito in *true*.

```
bool b = 5;  // 5 è diverso da 0, quindi b assume il valore true

int c = b;  // b è true, quindi c assume il valore 1
```


Tipo char

Una variabile di tipo *char* può contenere un carattere o un intero. I caratteri ammessi sono quelli definiti dallo standard ASCII, che comprende gran parte dei caratteri della tastiera.

```
char a = 'P';  // assegna il carattere 'P' alla variabile a

cout << a << " " << int(a) << endl;  // P 80

char b = 84;  // assegna l’intero 84 alla variabile b

cout << b << " " << int(b) << endl;  // T 84

b = b + 42;  // 126

cout << char(b) << endl;  // ~
```

In alcune implementazioni il tipo *char* è senza segno (come l’*unsigned char*). In tal caso è necessario utilizzare il tipo *signed char* per poter contenere i valori da -128 a 127.

Caratteri speciali

Carattere	Funzione
\n	Ritorno a capo
\t	Tabulazione orizzontale
\b	Cancellazione carattere precedente
\a	Allarme
\\	Barra
\"	Doppio apice
\0	Terminazione stringa

```
cout << "1\t2\t3\t4\t5\t6\n";
// 1    2    3    4    5    6

cout << "Corso di \"Programmazione Orientata agli Oggetti\"" << endl;
// Corso di "Programmazione Orientata agli Oggetti"
```


Costanti intere

I numeri interi possono essere rappresentati in notazione *decimale*, *esadecimale* e *ottale*. Un numero che inizia con **0x** è considerato esadecimale, mentre un numero che inizia per **0** è considerato ottale.

Esempi:

decimale:	**0**	**3**	**5**	**75**	**4788**
esadecimale:	**0x0**	**0x3**	**0x5**	**0x4b**	**0x12b4**
ottale:	**00**	**03**	**05**	**0113**	**011264**

E' possibile indicare esplicitamente il tipo di una costante intera utilizzando i suffissi **U** o **u** per indicare *unsigned* ed **L** o **l** per indicare *long*.

Esempi:

ottale 77 unsigned:	**077u**	**077U**		
decimale 3 unsigned long:	**3ul**	**3uL**	**3Ul**	**3UL**

Costanti in virgola mobile

Di default, un numero in virgola mobile è di tipo *double*. Per indicare esplicitamente che un numero è *float* si usa il suffisso **F** o **f**; per indicare che un numero è *long double* si usa il suffisso **L** o **l**.

Esempi di numeri *double*:

3.14 .99 1.0 1. 3.575e12 -3.575E-12

Esempi di numeri *float* e *long double*:

3.14F .99f 1.0L 1.l 3.575e12L -3.575E-12f

Lo spazio occupato da una costante numerica dipende dal tipo:

```
cout << sizeof(3.14) << endl;  // 8
cout << sizeof(3.14F) << endl;  // 4
cout << sizeof(3.575e12) << endl;  // 8
cout << sizeof(3.575e12L) << endl;  // 12
```


Il tipo long long

In alcuni sistemi è disponibile il tipo aggiuntivo *long long* (o *long long int*), utilizzato per rappresentare numeri interi con un intervallo più esteso del tipo *long*.

Su un sistema a 32 bit un *long long* ha generalmente una dimensione di 64 bit, ed è in grado di rappresentare i numeri interi appartenenti al seguente intervallo:

[-9223372036854775808, 9223372036854775807]

Il tipo senza segno *unsigned long long* (o *unsigned long long int*) può rappresentare i numeri interi appartenente all'intervallo:

[0, 18446744073709551615]

Il suffisso **LL** o **ll** può essere usato per indicare in modo esplicito che il tipo di un numero intero è *long long*.

Variabili e costanti

```
main () {
  int a;
  cout << a;  // attenzione: valore indeterminato
  a = 5;
  cout << a;  // 5
  double b = 3.141592;
  const int c = 10;
  c = 20;  // errore: modifica di una costante
  const int d = 3*4;  // ok
  const int e;  // errore: costante non inizializzata
}
```


Costanti in C

In C le costanti si dichiarano mediante la direttiva al preprocessore *define*:

```
#include <iostream>
#define N 3575
using namespace std;

int main() {
  cout << N + N << endl;  // 7150
  cout << N * 3 << endl;  // 10725
}
```

In questo esempio si definisce una costante senza tipo N uguale a 3575.

Il preprocessore sostituisce la costante N con il numero 3575 ovunque la trovi nel programma.

In C++ l'uso della direttiva *define* per dichiarare costanti, anche se ammesso, dovrebbe essere evitato.

Variabili globali

```
int a;  // variabile globale: inizializzata al valore di default 0
int b = 7;  // variabile globale inizializzata esplicitamente

main () {
  cout << a;  // 0
  b = b + 3;
  cout << b;  // 10
  int c;  // non inizializzata
  cout << c;  // valore indeterminato
}
```

Le variabili globali non inizializzate esplicitamente assumono il valore di default. Il valore di default delle variabili numeriche è 0. Il valore di default delle variabili booleane è *false*.

Operatori aritmetici

Operatore	Descrizione
–expr	meno unario
expr **+** expr	addizione
expr **–** expr	sottrazione
expr ***** expr	moltiplicazione
expr **/** expr	divisione
expr **%** expr	modulo
lvalue**++**	incremento postfisso
lvalue**--**	decremento postfisso
++lvalue	incremento prefisso
--lvalue	decremento prefisso

Operatori relazionali

<table>
<tr><th>Operatore</th><th>Descrizione</th></tr>
<tr><td>expr < expr</td><td>minore</td></tr>
<tr><td>expr <= expr</td><td>minore o uguale</td></tr>
<tr><td>expr > expr</td><td>maggiore</td></tr>
<tr><td>expr >= expr</td><td>maggiore o uguale</td></tr>
<tr><td>expr == expr</td><td>uguale</td></tr>
<tr><td>expr != expr</td><td>diverso</td></tr>
<tr><td>! expr</td><td>NOT</td></tr>
<tr><td>expr && expr</td><td>AND</td></tr>
<tr><td>expr || expr</td><td>OR</td></tr>
</table>

Operatori bit a bit

<table>
<tr><th>Operatore</th><th>Descrizione</th></tr>
<tr><td>expr & expr</td><td>AND bit a bit</td></tr>
<tr><td>expr | expr</td><td>OR bit a bit</td></tr>
<tr><td>expr ^ expr</td><td>XOR bit a bit</td></tr>
<tr><td>~expr</td><td>complemento</td></tr>
<tr><td>expr << expr</td><td>scorrimento a sinistra</td></tr>
<tr><td>expr >> expr</td><td>scorrimento a destra</td></tr>
</table>

```
unsigned short a = 3;        // 0000000000000011
unsigned short b = 17;       // 0000000000010001
unsigned short c = a | b;    // 0000000000010011
cout << c << endl;   // 19
unsigned short d = c << 1;   // 0000000000100110
cout << d << endl;   // 38
unsigned short e = c << 2;   // 0000000001001100
cout << e << endl;   // 76
```

Operatori di assegnamento

Operatore	Descrizione
lvalue **=** expr	assegnamento semplice
lvalue **+=** expr	addizione e assegnamento
lvalue **–=** expr	sottrazione e assegnamento
lvalue ***=** expr	moltiplicazione e assegnamento
lvalue **/=** expr	divisione e assegnamento
lvalue **%=** expr	modulo e assegnamento
lvalue **&=** expr	AND e assegnamento
lvalue **|=** expr	OR e assegnamento
lvalue **^=** expr	XOR e assegnamento
lvalue **~=** expr	complemento e assegnamento
lvalue **<<=** expr	scorrimento a sinistra e assegnamento
lvalue **>>=** expr	scorrimento a destra e assegnamento

Typedef

La parola chiave *typedef* consente di definire un nuovo nome per un tipo già esistente:

typedef int integer; // *integer* è un sinonimo di *int*

integer a = 5;

int b = 4+a;

Il meccanismo del *typedef* è impiegato anche per definire tipi “di sistema” utilizzati in molte funzioni e classi standard.
Per esempio un *size_type* è un intero senza segno utilizzato per le variabili contenenti dimensioni o indici:

typedef unsigned long size_type; // *size_type = unsigned long*

L’operatore *sizeof*, ad esempio, restitituisce un valore di tipo *size_type*.

Enumerazioni

Un'enumerazione è un tipo che può contenere un insieme di valori (detti enumeratori) specificati dall'utente. Si definisce mediante la parola chiave *enum*:

```
enum colore { rosso, verde, blu };  // rosso = 0, verde = 1, blu = 2

colore c = blu;

cout << c;  // 2
```

E' possibile attribuire un valore esplicito agli enumeratori:

```
enum mese { gennaio = 1, febbraio = 2, marzo = 3, aprile = 4 };

mese m1 = 3;  // errore: 3 non è di tipo mese

mese m2 = mese(3);  // ok
```


Array

Un array è un'aggregazione di elementi dello stesso tipo. In un array di dimensione *n* gli elementi sono indicizzato da 0 ad *n*-1.

```
int a[5];  // a è un array di 5 numeri interi

double b[10];  // b è un array di 10 numeri double

char c[2];  // c è un array di 2 caratteri
```

La dimensione dell'array deve essere nota a tempo di compilazione, in quanto solo alcuni compilatori supportano gli array a dimensione variabile.

Il seguente frammento di codice, anche se supportato da alcuni compilatori, dovrebbe quindi essere evitato per motivi di portabilità:

```
int dim;

cout << "Inserisci la dimensione dell'array: ";

cin >> dim;

int d[dim];  // attenzione: dimensione nota solo a tempo d'esecuzione
```

Array

Un array può essere inizializzato mediante una lista di valori:

```
int a[] = { 2, 4, 8, 16 };  // la dimensione viene calcolata automaticamente
int b[4] = { 2, 4, 8, 16 };  // ok
int c[4] = { 2, 4, 8, 16, 32 };  // errore: troppi valori
int d[4] = { 2, 4, 8 };  // ok: l'elemento mancante viene inizializzato a 0
```

Gli elementi dell'array possono essere acceduti utilizzando l'operatore []:

```
a[1] = a[0] + 3;
cout << a[1];
```

E' possibile dichiarare array multidimensionali:

```
const int n=3, m=4;
int e[n][m];
e[0][0] = 10;
```


Stringhe

Le stringhe sono generalmente gestite mediante la classe *string* della libreria standard.

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

int main() {
  string a = "Prova";
  string b = "stringhe";
  string c = a+" "+b;  // + è l'operatore di concatenazione
  cout << c << endl;  // Prova stringhe
  string s;
  cout << "Inserisci una stringa: ";
  cin >> s;  // legge una stringa fino al primo spazio e la memorizza in s
  cout << "Hai inserito: " << s << endl;  // stampa la stringa s
}
```

Stringhe

E’ possibile accedere ai singoli caratteri di un oggetto *string* mediante il metodo *at* o tramite l’operatore []:

```
string s = "Roma";
char a = s.at(2);  // a = 'm'
char b = s[3];  // b = 'a'
cout << a << "  " << b << endl;  // m  a
```

Per conoscere la lunghezza di una stringa si può usare il metodo *size* o il metodo *length*:

```
int lung = s.size();  // lung = 4
s[lung-1] = 'e';
cout << s << endl;  // Rome
```

Il metodo *empty* restituisce *true* se la stringa è vuota:

```
if (s.empty())
  cout << "stringa vuota";
```


Strutture

Una struttura è un’aggregazione di elementi di tipo arbitrario. Si definisce mediante la parola *struct*:

```
struct Automobile {
  string marca;
  string modello;
  int cilindrata;
};
```

E’ possibile dichiarare variabili di tipo *Automobile* come si dichiarano variabili di tipi predefiniti. Il *punto* è utilizzato per accedere ai singoli membri della struttura:

```
Automobile a;
a.marca = "Ferrari";
a.modello = "FXX";
a.cilindrata = 6262;
cout << a.marca << " " << a.modello << endl;  // Ferrari FXX
```

Nota: è preferibile fare uso delle *classi* anziché delle strutture.

Istruzione if-else

```
if (<condizione>)
  <blocco-if>
```

```
if (<condizione>)
  <blocco-if>
else
  <blocco-else>
```

La *condizione* è il risultato di una espressione booleana. Se il risultato è *true* si esegue il *blocco-if*, altrimenti si esegue il *blocco-else* (nel caso in cui sia stato specificato un ramo *else*).

Ogni blocco può contenere una o più istruzioni. Se un blocco contiene più di una istruzione deve essere racchiuso all'interno di una coppia di parentesi graffe.

Espressione condizionale

Un'*espressione condizionale* può essere usata in alcuni casi per sostituire un'istruzione *if-else*:

Per esempio l'istruzione:

```
if (n < 0)
  abs = -n;
else
  abs = n;
```

può essere sostituita dalla seguente espressione condizionale:

```
abs = (n < 0)  ?  -n  :  n;
```

Si noti che le parentesi intorno alla condizione possono essere omesse:

```
abs = n < 0  ?  -n  :  n;
```

Istruzione switch

L'istruzione *switch* può essere usata in alcuni casi per evitare l'uso di istruzioni *if-else* in cascata:

```
switch (val) {
 case val1:
  istruzioni
  break;
 case val2:
  istruzioni
  break;
 case val3:
  istruzioni
  break;
 default:
  istruzioni
}
```


Istruzioni di ciclo

```
for (<inizializzazione>; <condizione>; <espressione>)
 <blocco-istruzioni>
```

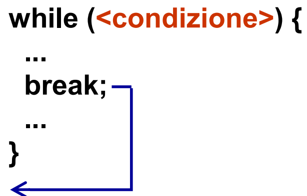
```
while (<condizione>)
 <blocco-istruzioni>
```

```
do
 <blocco-istruzioni>
while (<condizione>)
```

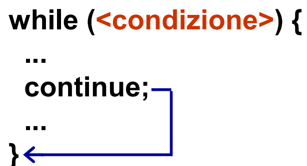
Come di consueto ogni blocco può contenere più istruzioni. In tal caso il blocco deve essere racchiuso all'interno di una coppia di parentesi graffe.

Istruzioni break e continue

L'istruzione *break* causa la terminazione del ciclo all'interno del quale è invocata.



L'istruzione *continue* determina il salto alla fine del blocco istruzioni del ciclo:



Si noti che nel caso di cicli innestati le istruzioni *break* e *continue* hanno effetto solo sul ciclo in cui sono contenuti.

Istruzione goto

L'istruzione *goto* determina il salto alla posizione identificata da una determinata *etichetta*. L'uso del *goto* di norma deve essere evitato. Un caso in cui il *goto* può essere utilizzato - nel caso in cui l'efficienza sia fondamentale - è la terminazione di una serie di cicli innestati da un punto del ciclo più interno.