Olimpiadi di Informatica 2011
Giornate preparatorie

Dipartimento di Informatica
Università di Torino

marzo 2011

1 – Dalla sintassi Pascal alla sintassi "alla *C*"

Dalla sintassi Pascal alla sintassi C/C++

Gran parte dei linguaggi di programmazione oggi più diffusi, ad esempio
C, C++, Java, C#, JavaScript
adotta una sintassi "alla C", o comunque più simile alla sintassi C che alla sintassi Pascal (vedi linguaggi come Python, Perl, ecc.).

La sintassi alla Pascal è quasi scomparsa.

Anche se i costrutti di base e i relativi concetti sono in gran parte gli stessi, la differenza fra Pascal e C non è puramente di sintassi superficiale: vi sono *anche* differenze concettuali.

Minuscole e maiuscole

C, C++, Java, C# sono CASE-SENSITIVE:

gli identificatori

`somma, Somma, SOMMA, somMA`

sono tutti diversi fra loro.

Nella maggior parte di tali linguaggi l'uso di maiuscole e minuscole negli identificatori, benché non vincolato in alcun modo dalla definizione del linguaggio, e quindi non forzato dal compilatore, deve tuttavia obbedire a precise regole di stile stabilite dalla comunità dei programmatori o dall'ente che ha definito il linguaggio.

In particolare, è considerato inaccettabile (benché il programma si compili correttamente!) usare identificatori tutti maiuscoli, tranne che nel caso delle costanti.

L'assegnazione

Simbolo per l' assegnazione ormai usato da quasi tutti i linguaggi moderni:

= invece di **:=**

Per i corsi di introduttivi di programmazione ciò crea problemi psicologici: uso di un simbolo matematico standard, graficamente simmetrico, denotante in matematica una relazione simmetrica fondamentale (l'ugua-glianza), per indicare un'operazione NON simmetrica!

Gli studenti tendono a confondere a = b con b = a: l'inconscio matematico!

Essendo il simbolo matematico di uguaglianza usato per l'assegnazione, l'uguaglianza è denotata da un simbolo diverso, che è (anch'esso quasi universalmente):

== invece di **=**

Il punto e virgola

Il punto e virgola non è un separatore di istruzioni come in Pascal, bensì
un terminatore(-costruttore) di istruzione e di dichiarazione
È parte integrante dell'istruzione, e va quindi posto obbligatoriamente anche dopo l'ultima istruzione di un blocco.

```
som = a + b;
dif = a - b;
```

La scrittura

```
som = a + b
```

NON è un'istruzione.

Dimenticare il punto e virgola di solito non crea problemi: i compilatori segnalano prontamente "errore: manca un punto e virgola".

Un punto e virgola da solo denota l'istruzione vuota; in conseguenza di ciò, come vedremo, mettere un punto e virgola nel posto sbagliato PUÒ creare problemi.

Dichiarazioni: il tipo prima della variabile (o parametro, ecc.)

Il nome del tipo precede il nome della variabile, parametro o altro ente che viene dichiarato. Non c'è la parola-chiave *var* né i due punti.

Esempio – dichiarazione di variabile:
Pascal: `var n: integer` diventa C, C++, C#, Java: `int n;`

Nota Bene: il punto e virgola finale è necessario, poiché è parte integrante della dichiarazione.

Esempio – dichiarazione di funzione (intestazione):
Pascal: function myFun(x: double; n: integer): double
C, C++, C#, Java: double myFun(double x, int n)

A differenza che in Pascal, le dichiarazioni in C++, Java, ecc. non devono stare tutte all'inizio, ma possono essere inframmezzate alle istruzioni:

```
a = a + b;
int i;
…
```

Dichiarazioni multiple.

Più dichiarazioni di variabili dello stesso tipo possono essere scritte in forma compatta usando la virgola:

```
int i, j, k;
```

è equivalente a:

```
int i;
int j;
int k;
```

La dichiarazione di una variabile può essere combinata con la sua inizializzazione (cioè la sua prima assegnazione): anzi, questo è lo stile considerato migliore.
Invece di scrivere:

```
int i;
int j;
i = 0;
j = m + n;
```

si può (anzi è meglio) scrivere:

```
int i = 0;
int j = m + n;
```

oppure, in modo ancora più compatto, ma forse meno chiaro:

```
int i = 0, j = m + n;
```


Le graffe

Un blocco di istruzioni non è delimitato da *begin end*, ma da parentesi graffe { }.
NOTA BENE: la coppia di graffe è un costruttore di istruzione (l'istruzione-blocco) come il punto e virgola, ma alternativo ad esso. Pertanto dopo la graffa di chiusura NON ci vuole il punto e virgola.

```
{
  a = a + b;
  b = a – b;
}
```

Problema

La vecchia tastiera italiana NON ha le graffe: bisogna pigiare ALT 123 e ALT 125. Nelle tastiere italiane più nuove le graffe si ottengono con combinazioni di tasti dipendenti dal tipo di tastiera
(ad es. ALTGR 7 e ALTGR 0).

Per questa ragione a Informatica abbiamo solo tastiere USA internazionali.

OSSERVA:

```
{
  a = a + b;
  b = a – b
}
```

errore di compilazione: ';' expected

```
{
  a = a + b;
  b = a – b;
};
```

si compila correttamente, ma contiene un'inutile istruzione vuota.

Anche il blocco vuoto { } è un'istruzione (in inglese *statement*) corretta.

L'assegnazione è anche un'espressione !

Il simbolo "=" è un operatore, e la forma (senza punto e virgola finale)

`a =` *espressioneADestra*

è a sua volta un'espressione che, oltre ad avere l'effetto di mettere in a il valore calcolato della *espressioneADestra*, ha essa stessa (cioè restituisce) tale valore.

Conseguenza: si può scrivere ad esempio

```
a = b = 10;
```

Tale scrittura viene interpretata come

```
a = (b = 10);
```

La valutazione dell'espressione `(b = 10)` ha l'effetto di depositare 10 in b; essa inoltre restituisce il valore 10 stesso, che viene – per effetto dell'assegnazione più esterna – depositato anche in a.
L'istruzione precedente è quindi equivalente alla sequenza di istruzioni

```
a = 10;
b = 10;
```

NOTA BENE: l'aggiunta di un punto e virgola finale trasforma una espressione di assegnazione in una istruzione di assegnazione.

Espressioni di assegnazione

Il fatto che l'assegnazione sia un'espressione permette di scrivere codice compatto ma criptico, ad esempio:

```
a = b * (c = 10);
```

In a viene messo il prodotto di b per 10, e contemporaneamente in c viene messo 10.

```
if ( (a = b) == 0) …
```

Nel test dell'if si testa se b è zero, e contemporaneamente si copia il contenuto di b in a.

Pratiche in generale sconsigliate nella programmazione odierna !
I programmi risultanti possono essere quasi incomprensibili !

Gli operatori booleani e relazionali più comuni

Pascal	C, C++, Java, ecc.
`and`	`&&`
`or`	`||`
`not`	`!`
`=`	`==`
`<>`	`!=`

Un operatore aritmetico: il resto della divisione intera

Pascal	C, C++, Java, ecc.
`mod`	`%`

Forme compatte di assegnazione

```
k++;        equivale a    k = k+1;
k--;        equivale a    k = k-1;

a += b;     equivale a    a = a + b;
a *= b;     equivale a    a = a * b;
a /= b;     equivale a    a = a/b;
a &&= b;    equivale a    a = a && b;
```

eccetera, per tutti gli operatori binari aritmetici e logici.

È buona norma usare gli operatori di incremento e decremento `++` e `--` (che sono stati applicati anche ai nomi dei linguaggi, generando il … C++) invece delle corrispondenti forme prolisse alla Pascal.
Ma attenzione:

l'espressione `k++` NON è equivalente all'espressione `k+1`.

Input-output

Nei linguaggi moderni, a differenza che in TurboPascal, la definizione del linguaggio non comprende mai delle primitive di input-output, neppure per l'input-output su consolle (cioè su finestra a righe di comando alla DOS).

L'input-output si effettua attraverso procedure di libreria, che hanno nomi e caratteristiche diverse da linguaggio a linguaggio.
Ad esempio, per quanto riguarda l'output su consolle, l'istruzione che scrive "Ciao a tutti", da inserire opportunamente nel solito primo programma, è:

C puro:
```
printf("Ciao a tutti\n");
```

C++:
```
cout << "Ciao a tutti\n";
```

Java:
```
System.out.println("Ciao a tutti");
```

C#:
```
System.Console.WriteLine("Ciao a tutti");
```

Il blocco e le dichiarazioni

Un blocco può contenere sia istruzioni che dichiarazioni; a differenza che in Pascal, il corpo di una procedura o funzione è semplicemente un blocco, e le dichiarazioni locali devono stare all'interno di tale blocco.

Pascal:

```
function fattoriale(n: integer): integer;
  var ris, i: integer;
begin
  ris = 1;
  for i:= 2 to n do ris = ris * i;
  fattoriale = ris
end;
```

C, C++, ecc.:

```
int fattoriale(int n) {
  int ris = 1;
  for(int i = 2; i < n; i++) ris *= i;
  return ris;
}
```

(per gli altri aspetti della sintassi che compaiono qui si veda più avanti)

Non ci sono procedure, ma solo funzioni!

Una *procedure* nel senso del Pascal è semplicemente una funzione in cui il tipo del risultato è lo speciale tipo void, che non contiene alcun valore.

Pascal:

```
procedure ciao(string nome);
begin
  write('Ciao, ');
  writeln(nome);
end;
```

C++:

```
void ciao(string nome) {
  cout << "Ciao, " + nome << endl;
}
```

Non ci sono funzioni definite dentro funzioni!

Non si possono definire funzioni o procedure all'interno di altre funzioni o procedure. Tutte le funzioni sono quindi allo stesso livello.

Sintatticamente non esiste neppure un programma principale all'interno del quale siano definite le procedure o funzioni.

Un programma C o C++ è semplicemente un insieme di dichiarazioni di variabili (le variabili globali) e di funzioni.
Una delle funzioni si deve chiamare main, e l'esecuzione del programma comincia automaticamente con la chiamata di tale funzione: essa poi in generale richiamerà le altre funzioni, ecc.
Le variabili dichiarate dentro il main sono quindi variabili locali del main, e non sono accessibili dalle altre funzioni.

Nota per programmatori provenienti dal Pascal

NOTA BENE: Non si possono scrivere istruzioni (salvo le inizializzazioni eventualmente contenute nelle dichiarazioni) al top-level: le istruzioni possono stare solo all'interno di una funzione, eventualmente la funzione main.

Esempio.
Il solito primo programma che scrive "Ciao a tutti" sulla consolle non può essere scritto semplicemente (in C++):

```
cout << "Ciao a tutti“;
```

bisogna scrivere invece almeno (in C++):

```
#include <iostream.h>
#include <stdlib.h>

void main() {
  cout << "Ciao a tutti\n";
  system("PAUSE"); // solo su DevC++ per Windows
}
```

Osserva:

Diversamente dal Pascal, nella definizione e nell' invocazione di funzioni senza argomenti occorre sempre scrivere le parentesi.

Esempio (C++):

```
#include <iostream.h>
#include <stdlib.h>

void ciao() {
  cout << "Ciao a tutti!" << endl;
}

void main() {
  ...
  ciao();
  system("PAUSE");
}
```


I costrutti di base: l'if e l'if-else.

Il costrutto if

if(*Espressione_Booleana* **)** *Istruzione*

Esempio

```
if(n > 0)
  n = fattoriale(n);
```

istruzione semplice

NOTA BENE:

- le parentesi intorno alla condizione fanno parte integrante della sintassi dell'*if* e sono quindi obbligatorie;
- non si scrive la parola *then* (la separazione fra la condizione e il successivo statement è assicurata dalla parentesi);

```
if(n > 0) {
  ris1 = fattoriale(n);
  ris2 = potenza(2,n);
}
```

istruzione-blocco

Il costrutto if-else

if(*Espressione_Booleana* **)** *Istruzione*
else *Istruzione*

Esempi

```
if(n > 0)
  n = fattoriale(n); // Nota: punto e virgola obbligatorio !
else
  n = -n;

if(n > 0) {
  ris1 = fattoriale(n);
  ris2 = potenza(2,n);
}         // RICORDA: dopo la chiusa-graffa NON ci vuole il ";" !
else n++;
```


if-else annidati
e "istruzione virtuale" if ... else if ... else if ... else ...

if(*espress-booleana*$_1$ **)**
  istruzione$_1$
else if(*espress-booleana*$_2$ **)**
  istruzione$_2$
else if(*espress-booleana*$_3$ **)**
  istruzione$_3$
...
else
  istruzione$_n$
}

Istruzione virtuale if ... else if ... else if ... else ...

Vengono valutate in sequenza le condizioni
espress-booleana$_1$
espress-booleana$_2$
...
finché si trova una condizione *espress-booleana*$_i$ che vale *true*: si esegue la corrispondente *istruzione*$_i$ e poi si "saltano" tutte le istruzioni corrispondenti alle condizioni successive.
Se nessuna delle *espress-booleana*$_i$ vale *true,* si esegue l'istruzione corrispondente alla condizione "pigliatutto" *else*.

I costrutti di base: il while

while(*Espressione_Booleana* **)** *Istruzione*

NOTA BENE: le parentesi intorno alla condizione fanno parte integrante della sintassi del while e sono quindi obbligatorie

Esempio 1

```
int prodotto = 5;
while(prodotto <= 100)
  prodotto = 3*prodotto; // istruzione semplice
```

Esempio 2

```
double x = 2.1;
int n = 18;
ris = 1;
while(n > 0) {          // istruzione-blocco
  if(n%2 == 0) {
    x = x*x;
    n = n/2;
  }
  else {
    ris = ris*x;
    n--;
  }
}
```

Attenzione ai punti e virgola nei posti sbagliati!

Il seguente pezzo di programma è sintatticamente corretto (quindi non genera errori di compilazione):

```
int x = 20;
while(x < 100);
  x = x + 6;
```

Ma l'istruzione while non termina, poiché il suo corpo NON è l'istruzione "`x = x + 6;`" bensì l'istruzione vuota costituita dal punto-e-virgola, la quale ovviamente non modifica x e quindi non renderà mai *false* la " condizione.

Il programma è logicamente errato!

Nell'if e nel while non mettere mai un punto e virgola dopo la condizione!

I costrutti di base: il do-while

Corrisponde al repeat del Pascal, ma a differenza di esso si esce dal ciclo per false (come nel while) e non per true !

```
do
  Istruzione
while(Espressione_booleana );
```

Nota: *Istruzione* può essere un blocco; essa viene chiamata *corpo* del do.
NOTARE che, AL CONTRARIO che nel ciclo while,
il punto e virgola finale non solo non è sbagliato, ma è obbligatorio.

Esecuzione:

1. si esegue il corpo;
2. si valuta l' *Espressione_booleana*:
   - false: si esce dal ciclo;
   - true: si torna al punto 1.

Inserire qui il diagramma di flusso

Nota

Il corpo del **do** viene eseguito **almeno una volta**.

Nota

La sintassi alla C dei cicli è un po' infelice. Soprattutto quando guardate un programma complesso, attenti a non confondere:

il `while` alla fine di un'istruzione **do-while**

con

il `while` all'inizio di un'istruzione **while**

I costrutti di base: il for

A differenza che in Pascal, nei linguaggi alla C non esiste un vero for (cioè un'istruzione iterativa in cui il numero di iterazioni sia garantito essere un numero finito). Il for è semplicemente un while scritto in modo più compatto.

```
for(iniz; condizione; incr)
  istruzione
```

è rigorosamente equivalente a:

```
iniz;
while(condizione) {
  istruzione;
  incr;
}
```

In un'istruzione for:

l'inizializzazione *iniz* può essere una dichiarazione con inizializzazione, ad esempio:

```
for(int i = 0; ... ; ...)
```

incr può essere qualunque istruzione semplice (senza il punto-e-virgola finale), ma un buono stile richiede che sia una istruzione di incremento o decremento;
ad esempio (Java):

```
for(int i = 0; i < 10; i++)  out.println(i);
for(int i = 10; i > 0; i--) out.println(i);
for(int i = 0; i < 10; i += 3)  out.println(i);
```

ecc.

Nota di stile

Come abbiamo visto, quando il corpo di un'istruzione di selezione o di iterazione è costituito da una sola istruzione, non è necessario racchiuderla in un blocco.
Tuttavia molti preferiscono, per uniformità, racchiudere in ogni caso il corpo dell'istruzione fra parentesi graffe.

Esempio:

```
for(int i = 1; i < n; i++)
  ris = ris*i;
```

oppure

```
for(int i = 1; i < n; i++) {
  ris = ris*i;
}
```

Nella seconda forma, se successivamente si deve correggere o modificare il programma aggiungendo delle istruzioni al corpo del for, non si rischia di dimenticare di inserire le graffe.

Altra nota per i programmatori provenienti dal Pascal

Nei linguaggi alla C non si può usare il nome di una funzione come pseudo-variabile contenente il risultato. Si tratta di una caratteristica molto discutibile del Pascal; nei linguaggi alla C per restituire il risultato di una funzione si usa la molto più chiara e semplice istruzione return.

Pascal:

```
function f(x: integer; y: integer): integer;
begin
  f = 3*x - 2*y
end;
```

C, C++, ecc.:

```
int f(int x, int y) {
  return 3*x - 2*y;
}
```

(notare il punto e virgola finale obbligatorio)

L'istruzione return

Vi sono due possibili forme dell'istruzione return:

`return;` oppure **`return`** *Espressione;*

In entrambi i casi, l'esecuzione dell' istruzione fa terminare l'esecuzione della funzione e restituisce il controllo alla funzione chiamante (o, nel caso del main, al sistema operativo).

La forma **`return`** *Espr;*
inoltre, calcola il valore dell'espressione *Espr* e lo "restituisce" (cioè lo passa "all'indietro") come risultato al chiamante.

NOTA BENE

L'istruzione **`return`** *Espr ;* può comparire solo in una funzione in cui il tipo del risultato sia uguale al tipo di *Espr*
L'istruzione **`return;`** può comparire solo in una funzione in cui il tipo del risultato sia void.

OSSERVA:

(Java)

```
void saluta() {
  return;
  System.out.println("Ciao");
}
```

L'istruzione

```
System.out.println("Ciao");
```

non verrebbe mai eseguita, poiché successiva ad una return che viene sempre eseguita.
Il compilatore Java "si accorge" di ciò, e non compila; genera invece il messaggio di errore:

unreachable statement

cioè "istruzione irraggiungibile".

Una funzione analoga in C o C++ viene invece da molti compilatori compilata senza errore.

L'istruzione return: altri esempi

```
int  f(int x) {
    return 3.0*x + 2.0;
 }
```

ERRORE di COMPILAZIONE! la return restituisce un double, ma la funzione deve restituire un int.

Osserva però:

```
double  f(int x) {
  return 3*x + 2;
}
```

è corretto, perché gl'interi possono essere considerati un sottoinsieme dei double (in realtà, la spiegazione formale precisa è un po' più complessa, ma qui questa può bastare)

Istruzione return posta dentro un ciclo

L'esecuzione di una return posta all'interno di un ciclo, facendo uscire dal metodo, interrompe necessariamente il ciclo:

(C++)

```
...
 bool ricerca(int x, int a[], int n) {
   for(int i=0; i < n; i++) {
     if(a[i] == x) return true;
   }
   return false;
 }
```

Se un elemento dell'array è uguale all'elemento cercato x, viene eseguita la return all'interno del ciclo, che così viene interrotto; quindi la return false dell'ultima riga, che è fuori dal ciclo, in tal caso non viene eseguita.

Se invece nell'array non vi sono elementi uguali a x, la return true non viene eseguita, quindi il for termina normalmente e poi viene eseguita la successiva istruzione return false.

Stili di scrittura dei blocchi

Stile 1 (più compatto)

```
double exp(double x, int n) {
  double ris = 1;
  while(n > 0) {
    if(n%2 == 0) {
      x = x*x;
      n = n/2;
    }
    else {
      ris = x*ris;
      n--;
    }
  } // fine del while
  return ris;
}
```

Stile 2 (graffe corrispondenti allineate)

```
double exp(double x, int n)
{
  double ris = 1;
  while(n > 0)
  {
    if(n%2 == 0)
    {
      x = x*x;
      n = n/2;
    }
    else
    {
      ris = x*ris;
      n--;
    }
  }
  return ris;
}
```


Arrays

Pascal
```
var a: array[1..20] of integer;
```

C, C++:
```
int a[20];
```

oppure, meglio:

```
int n = 20;
int a[n];
```

Attenzione: gli array sono indiciati a partire da 0

In tutti i linguaggi con sintassi alla C gli array sono indiciati a partire da 0:
gli elementi di un array ar di lunghezza 20 sono ar[0], ar[1], …, ar[19];
il ciclo tipico di scansione di un array di lunghezza n è quindi della forma:

```
for(int i = 0; i < n; i++) …
```

Array di lunghezza (fissa) determinata a runtime

```
int n;

int main() {
  printf("immetti un intero: ");
  scanf("%d", &n);
  int a[n];
  int i;
  for(i = 0; i < n; i++ ) {
    a[i] = i*i;
  }

  for(i = 0; i < n; i++ ) {
    printf("%d ", a[i]);
  }
  return 0;
}
```


L'espressione condizionale

È un costrutto che non esiste in Pascal, e che esiste invece nei linguaggi cosiddetti *funzionali*. È una specie di if-then-else che però è un'espressione invece che un istruzione. Sintassi:

Espressione_Booleana **?** *Espressione*$_1$ **:** *Espressione*$_2$

Il valore dell'espressione complessiva è*:*
• il valore di *Espressione*$_1$ se *Espressione_Booleana* vale *true*,
• il valore di *Espressione*$_2$ se *Espressione_Booleana* vale *false*.

Naturalmente *Espressione*$_1$ ed *Espressione*$_2$ devono avere lo stesso tipo.
Esempio:

```
max = a > b ? a : b;
```

in max viene messo il maggiore dei valori di a e b.
Equivale a:

```
if(a > b) max = a;
else max = b;
```

Allocazione dinamica: liste concatenate.

```
#include <stdio.h>
#include <stdlib.h>
```

definisco il tipo-struttura di nome "*struct node*", avente due campi:
- un campo-dati *element*, che per semplicità supponiamo sia di tipo int;
- un campo *next* di tipo puntatore a una struttura dello stesso tipo .

```
struct node {int element; struct node* next;};
```
per comodità e per chiarezza concettuale definiamo il nome *intlist* come sinonimo del tipo "*struct node **", cioè puntatore a nodo.
```
typedef struct node* intlist;
```

In modo più elegante:

```
typedef struct node * intlist;
struct node {int element; intlist next;};
```


Creazione di un nodo

funzioni (procedure):
costruisce un nuovo nodo con il dato e con il puntatore al successivo che le vengono passati, e restituisce il puntatore al nuovo nodo:

```
intlist newList(int elem, intlist lis) {
  intlist pnode = malloc(sizeof(struct node));
  pnode->element = elem;
  pnode->next = lis;
  return pnode;
}
```

costruisce un nuovo nodo con successivo NULL (ultimo nodo):

```
intlist newList1(int elem) {
  return newList(elem, NULL);
}
```

I/O di liste da console

lettura di una sequenza da consolle e costruzione della lista, ricorsiva (costruisce la lista e ne restituisce il puntatore al primo nodo):

```
intlist readList() {
  int x;
  scanf("%d", &x);
  return feof(stdin) ? NULL : newList(x, readList());
}
```

la versione iterativa è più difficile da scrivere!

I/O di liste da console

scrittura di una lista sullo schermo, ricorsiva:

```
void printList(intlist l) {
  if(l) {
    printf("%d ", l->element);
    printList(l->next);
  }
  else printf("\n");
}
```

scrittura di una lista sullo schermo, iterativa (più efficiente):

```
void printListIt(intlist l) {
  while(l) {
    printf("%d ", l->element);
    l = l->next;
  }
  printf("\n");
}
```

Esempio di programma con liste

```
int main() {
  intlist myList = readList();
  printList(myList);
  intlist cursor = myList;
  while(cursor) {
     cursor->element *= 2;
     cursor = cursor -> next;
  }
  printListIt(myList);
  return 0;
}
```


Conclusioni

La sintassi illustrata è comune a tutti i linguaggi moderni “alla C”.

I costrutti per la programmazione a oggetti, invece, sono sintatticamente e semanticamente diversi da linguaggio a linguaggio (e inesistenti in C puro), benché i concetti di base siano simili.
Essi verranno quindi presentati in uno specifico linguaggio.