Corso di

Linguaggi di Programmazione

- Laboratorio -

Stefano Ferilli

ferilli@di.uniba.it

Università degli Studi di Bari

A.A. 2001/2002

Introduzione al linguaggio C

Corso di Linguaggi di Programmazione

Università degli Studi di Bari

A.A. 2001/2002

Il linguaggio C

- Tendente al basso livello
  - Corrispondenza con le istruzioni macchina
  - Efficienza
- Standard ANSI
  - Portabilità
- Tipizzazione debole
  - Controlli poco rigorosi
  - Comportamenti imprevisti

Testi consigliati

- Linguaggi di programmazione
  - R. Sethi (Zanichelli)
- Linguaggio C (ANSI C) – II ed.
  - B.W. Kernighan, D.M. Ritchie (Jackson)
- C Corso Completo di Programmazione
  - H.M. Deitel, P.J. Deitel (Apogeo)

Prerequisiti - Obiettivi

- Prerequisiti
  - Programmazione strutturata
    - Teorema di Bohm-Jacopini
  - Pascal
    - Confronto fra i due linguaggi
- Obiettivi
  - Operatività in tempi ridotti
    - Approfondimenti successivi

Scaletta

- Struttura di un programma
- Funzioni
- Tipi semplici e modificatori
- Assegnamento ed operatori aritmetici
- Operatori relazionali e logici
- Puntatori
- I/O base
- Strutture di controllo
- Tipi strutturati
- Introduzione all'ambiente Borland C++

Legenda

- Pascal
  - Caratteri ombreggiati
- C
  - Caratteri in nero
- Il C è case-sensitive
  - Parole chiave minuscole
- Caratteri:
  - In tondo
    - A scelta del programmatore
      - Es.: identificatori
  - In **grassetto**
    - Definiti dalla sintassi
      - Es.: parole chiave
  - In *corsivo*
    - A scelta fra valori dati
      - Es.: tipi

Struttura di un programma

- Main **program**
  - Procedure
    - Procedure
      - …
    - Funzioni
      - …
  - Funzioni
    - Procedure
      - …
    - Funzioni
      - …

- Funzione **main**
- Funzione
- …
- Funzione

Struttura di un programma

- Nidificata
  - Astrazione funzionale
- Sezione dichiarativa
  - Visibilità: l'intera procedura/funzione
  - Leggibilità, controllo

- Appiattita
  - Funzioni paritarie
- Dichiarazioni sparse
  - Visibilità: il blocco in cui sono dichiarate
  - Efficienza in spazio

Funzioni

function nome**(**[**var**] arg**:** *tipo*,…, [**var**] arg**:** *tipo***):** *tipo***;**

dichiarazioni

begin … **end;**

tipo nome**(***tipo* arg, …, *tipo* arg**)**

/* dichiarazioni nel corpo */

{ … **}**

- Passaggio parametri
  - riferimento o valore
- Valore di ritorno
  - nome **:=** *valore*

- Passaggio parametri
  - valore
- Valore di ritorno
  - **return** *valore***;**

Il più semplice programma C

```
main () {}
```

abbreviazione per:

```
int main (void) {}
```

- Le funzioni restituiscono per default un intero

void

- Parola chiave per specificare l’assenza di parametri o di risultato

Tipi semplici

- **integer, boolean**
- **real**
- **char**

- **int, short, long**
- **double, float**
- **char**

- boolean $\rightarrow$ int
  - Falso $\rightarrow$ 0
  - Vero $\rightarrow$ $n \neq 0$

- char $\equiv$ int (byte)
  - Caratteri stampabili >0
- float usato solo per economia di spazio

Modificatori di tipo

- Ampiezza

  char (8 bit) < short (16 bit) < int < long (32 bit)

  float < double < long double

  - Possono coincidere con 1, 2 o 3 ampiezze diverse
- Segno

  signed/unsigned

  - Applicabili a char e int
  - Estensione del segno

Il pre-processore

- Direttive

```
#include "nomefile"
#include <nomeheader.h>
```

  - Inclusione di file esterni
    - I file di intestazioni raccolgono dichiarazioni di costanti e funzioni definite in altri moduli

`#define NOME` *`espansione`*

  - Definizione di costanti simboliche
    - Prima della compilazione tutte le occorrenze di `NOME` vengono sostituite da *`espansione`*

Variabili - Costanti

var id: *tipo*;

- Dichiarazione obbligatoria
  - Globali se dichiarate nel programma principale
  - Solo dichiarazione

tipo id **[=** *valore***];**

- Dichiarazione obbligatoria
  - Globali se dichiarate fuori dalle funzioni
  - Possibile inizializzazione

const id **=** *valore*;

- Costanti di piattaforma: `<limits.h>`

const *tipo* id **=** *valore***;**

- Costanti matematiche: `<float.h>`

Costanti enumerative

enum nome **{** ID [**=** *valore*]**,** …**,** ID [**=** *valore*] **};**

- Valori non necessariamente distinti
  - Valori progressivi dall'ultimo specificato
  - Se non specificato, il primo valore è 0
- Nomi diversi in enumerazioni distinte

Istruzioni

- **;** separatore
- Funzioni elementari predefinite
  - I/O di base
  - matematiche
- **(*** … ***)**, **{** … **}**

- **;** terminatore
- Nessuna funzione predefinita

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

- **/*** … ***/**

Assegnamento
Operatori aritmetici

- **:=**

  var **:=** var *op espr*

  var **:=** var **+ 1**

  var **:=** var **- 1**

- **+**, **-**, *

- **/**, **div**

- **mod**

- **=**

  var *op*= *espr*

  var**++** **++**var

  var**--** **--**var

- **+**, **-**, *

- **/**

- **%**

Operatori relazionali
Operatori logici

- =
- <>
- <, <=, >, >=

- not
- and
- or

- ==
- !=
- <, <=, >, >=

- !
- &&
- ||

Puntatori

&variabile

- Indirizzo in memoria di variabile
  - Utilizzabile per il passaggio di parametri per riferimento

*puntatore

- Valore contenuto all'indirizzo di memoria puntatore
  - ***** operatore di "indirezione"

tipo *puntatore

- *puntatore è di tipo *tipo*, ossia
  puntatore è l'indirizzo di un valore di tipo *tipo*

Puntatori - Esempio

```
int x = 1, y = 2, z[10];
int *ip;  /* ip puntatore ad un intero */
ip = &x;  /* ip punta ad x */
y = *ip;  /* y vale 1 */
*ip = 0;  /* x vale 0 */
ip = &z[0]/* ip punta a z[0] */
```

- Esempi di costrutti validi:

```
*ip = *ip + 10;         y = *ip + 1;
*ip += 1;    ++*ip;     (*ip)++
```

Procedura di scambio di valori

```
void swap(int *x, int *y) {
  int temp;

  temp = *x;
  *x = *y;
  *y = temp;
  }
```

- Viene chiamata con i riferimenti: `swap(&a, &b);`

I/O base

- write(…)
  - writeln(…)
- read(…)

- printf("…", …)
  - printf("…\n", …)
- scanf("…", …)

Funzioni di I/O base

printf(“*formato***”,** parametro**, …,** parametro**)**

scanf(“*formato***”, &**parametro**, …, &**parametro**)**

- formato è la stringa da stampare/leggere
  - Possibilità di inserire caratteri speciali
  - Segnaposto per i parametri da stampare/leggere
    - Corrispondenza fra numero e tipo dei segnaposto e sequenza dei parametri
  - Numero di spaziature ignorato in lettura

Caratteri speciali

- Sequenze di escape
  - \n a capo
  - \t tab orizzontale
  - \v tab verticale
  - \b cancellazione
  - \” doppi apici
  - \’ apice singolo
  - \? punto interrogativo
  - \\ barra rovesciata
- Segnaposto parametri
  - %d decimale
  - %i intero
  - %f virgola mobile
  - %e esponenziale
  - %c carattere
  - %s stringa
  - %% stampa ‘%’

Programma di benvenuto

```
#include <stdio.h>
main () {
printf(“Salve, mondo\n”);
}
```

```
#include <stdio.h>
main () {
  printf(“Salve, ”);
  printf(“mondo”);
  printf(“\n”);
  }
```

Somma e media di due valori

```
#include <stdio.h>
main() {
   int a,b,s;

   printf("Inserisci due numeri interi:\n");
   scanf("%d %d",&a,&b);
   s = a + b;
   printf("Somma = %d\tMedia = %f\n",s,s/2.0);
   }
```

Sequenza

begin … **end**[;]

{ … **}**

- Località delle variabili
  - Non inizializzate

Selezione binaria

if *condizione* **then**
istruzione
[**else**
istruzione]

if (*condizione***)**
istruzione
[**else**
istruzione]

Parità di un intero

```
#include <stdio.h>

main() {
   int n;

   printf("Inserisci un intero: ");
   scanf("%d",&n);
   if (n % 2 == 0)
       printf("%d e' pari\n",n);
   else
       printf("%d e' dispari\n",n);
   }
```

Selezione multipla

case espressione **of**
valori: *istruzione*;
…
valori: *istruzione*[;
otherwise *istruzione*]
end;

switch (*espressione***)**
{
case *costante*: *istruzione*
[**break;**]
…
case *costante*: *istruzione*
[**break;**]
default: *istruzione*
}

Modulo 3

```
#include <stdio.h>
main() {
   int n, r;
   printf("Inserisci un intero: ");
   scanf("%d",&n);
   switch (r = n % 3) {
       case 1: printf(“%d mod 3 = %d\n",n,r);
       break;
       case 2: printf("%d mod 3 = %d\n",n,r);
       break;
       default: printf("%d mod 3 = 0\n",n);
       }
   }
```

Selezione in cascata

- Esecuzione in cascata
  - Evitabile con **break**
- Ammessi solo valori costanti singoli

if (*condizione***)**
    istruzione
else if (*condizione***)**
    istruzione
…
else if (*condizione***)**
    istruzione
[**else**
    istruzione]

Segno di un intero

```
#include <stdio.h>

main() {
   signed int n;

   printf("Inserisci un intero: ");
   scanf("%d",&n);
   if (n > 0)
       printf("Positivo\n");
   else if (n < 0)
       printf("Negativo\n");
   else
       printf("Nullo\n");
   }
```

Iterazione

while *condizione* **do**
    istruzione

while (*condizione***)**
    istruzione

repeat
    istruzione
until *condizione-falsa*

do
    istruzione
while *condizione-vera*

Conto alla rovescia

```
#include <stdio.h>

main() {
   int n = 10;

   while (n != 0)
       {
       printf("%d\n",n);
       n--;
       }
   printf("Contatto!\n");
   }
```

Inserimento di valori negativi

```
#include <stdio.h>

main() {
   signed int n;

   do {
       printf("Inserisci un intero negativo: ");
       scanf("%d",&n);
       }
   while (n >= 0);
   }
```

Iterazione

for *ind* **:=** *inf* **to** *sup* **do**
    istruzione;

≡

ind **:=** *inf*;
while *ind* **<=** *sup* **do**
    begin
        istruzione;
        ind **:=** *ind* **+ 1**
    end;

for (*espr1***;** *espr2***;** *espr3***)**
    istruzione

≡

espr1;
while *espr2*
    {
        *istruzione***;**
        *espr3***;**
    }

Calcolo dei primi *n* numeri dispari

```
#include <stdio.h>
main() {
  int n;

  printf("Quanti numeri devo
  generare?\n");
  scanf("%d",&n);
  for(int i=0; i<n; i++)
     printf("%6d.%10d\n",i+1,2*i+1);
  }
```

Vettori

var:
    array[*dim***] of** *tipo*;

- Indici enumerativi
  - Intervallo a piacere

- Accesso:
  variabile**[**ind**]**

tipo var**[**[*dim*]**]**
    [**= {***val***,** …**,** *val***}**]**;**

- Indici interi
  - Partono sempre da 0
- *var* ≡ **&***var***[**0**]**

- Accesso:
  variabile**[**ind**]**

Vettori multidimensionali

var**:**
array[*dim***,**…**,***dim***]**
of *tipo*

- Accesso:

var**[**ind**,**…**,**ind**]**

tipo var**[**[*dim*]**]**…**[**[*dim*]**]**
[**= {{***val***,** …**,** *val***},** …**,**
{*val***,** …**,** *val***}}**]

- Memorizzati per righe

- Accesso:

var**[**ind**]**…**[**ind**]**

Structure

variabile**: record**
id**:** *tipo*;
…
id**:** *tipo*;
end;

- Accesso:

  variabile**.**id

struct [nome] **{**
tipo id**;**
…
tipo id**;**
} [variabile **=** **{**val**,** **…,** val**}**]**;**

- Accesso:

  variabile**.**id

Strutture Dinamiche

Corso di Linguaggi di Programmazione

Università degli Studi di Bari

A.A. 2001/2002

Strutture

p
- Puntatore a struttura

*p
- Contenuto della struttura

(*p).el abbreviato in p->el
- Elemento della struttura

Definizione di nuovi tipi

typedef *tipo* Nome;

- Semplici sinonimi di tipi esistenti
  - Possibilità di dare un nome a tipi complessi
- Interpretata dal compilatore (`#define` no)
  - Parametrizzazione di programmi (portabilità)
  - Significatività dei nomi (documentabilità)

Puntatori

var puntatore**:** **^***tipo***;**	*tipo* *****puntatore**;**
puntatore**^**	*****puntatore
puntatore	puntatore
nil	**NULL**

Puntatori

NULL

- Puntatore nullo
- Costante simbolica definita uguale a 0
  - Utilizzabile per il test booleani
  - 0 non può essere un indirizzo di memoria valido

Forzatura di tipi

(*tipo***)** *espressione*

- Operatore di “Cast”
- Forzatura di un valore ad un tipo
  - Il tipo deve essere compatibile col valore

Allocazione dinamica

var puntatore**: ^***tipo***;**	*tipo* *****puntatore**;**
new(puntatore**)**	puntatore **= (***tipo* *****)** **malloc(***dim***);**
dispose(puntatore**)**	**free(**puntatore**);**
nil	**NULL**

Allocazione dinamica

void *malloc(size_t n_bytes**)**

- Definita in `<stdlib.h>`
- Valori di ritorno:
  - Indirizzo per un oggetto del tipo richiesto
  - **NULL** se lo spazio non è disponibile
- Cast obbligatorio per il tipo richiesto
- Richiede il numero di byte da allocare
  - Calcolabile mediante l'operatore **sizeof()**

Allocazione dinamica

size_t sizeof(*oggetto***)**

size_t sizeof(*tipo***)**

- Restituisce la dimensione di oggetto o tipo

  typedef size_t = unsigned int

  - Definito in `<stddef.h>`

void free(void ***puntatore)**

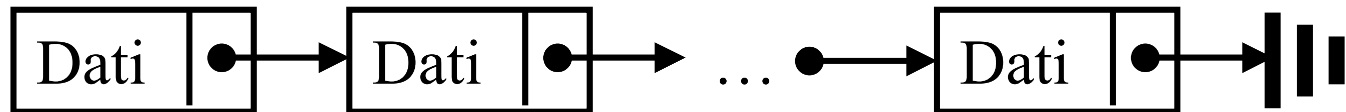
- Libera la memoria dell'oggetto in puntatore
  - Deve essere stata allocata dinamicamente

Strutture dinamiche

- Elementi omogenei (= dello stesso tipo)
- Residenti in memoria centrale
- Numero di elementi non fissato
  - Aggiunta secondo necessità
    - Allocazione dinamica della memoria
  - Accesso attraverso elementi precedenti
    - Elemento =
Dati + Puntatori agli elementi seguenti

Liste concatenate

- Sequenze lineari di elementi
  - Accesso tramite scansione sequenziale
    - Ciascun elemento punta al successivo
    - L’ultimo elemento ha un terminatore



Liste concatenate

```
struct el_lista {
… /* dati */
struct el_lista *el_succ;
} *lista = NULL;

struct el_lista *new_el =
    (struct el_lista *)malloc(sizeof(struct el_lista));
(*elem).el_succ = new_el;  ≡ elem->el_succ = new_el;
(*new_el).el_succ = NULL;≡ new_el->el_succ = NULL;
```

Acquisizione di una lista di interi (I)

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

struct el_lista {
   int dato;
   struct el_lista *el_succ;
   };
typedef struct el_lista *Elem;
int stampa_lista(Elem);
Elem add_el(void);
```

Acquisizione di una lista di interi (II)

```
main() {
   Elem lista = NULL, new_el, last_el;

do {
      if ((new_el = add_el()) != NULL) {
         if (lista == NULL)
            lista = new_el;
         else
            (*last_el).el_succ = new_el;
         last_el = new_el;
         }
   } while (new_el != NULL);
   stampa_lista(lista);
}
```

Acquisizione di una lista di interi (III)

```
Elem add_el(void) {
   int i; Elem new_el;

   printf("Inserisci un intero positivo: ");
   scanf("\n%i",&i);
   if (i > 0) {
      new_el = (Elem)malloc(sizeof(struct el_lista));
      (*new_el).dato = i;
      (*new_el).el_succ = NULL;
      return new_el;
      }
      else
         return NULL;
   }
```

Acquisizione di una lista di interi (IV)

```
int stampa_lista(Elem last_el) {
    while(last_el != NULL) {
        printf("%i\n",(*last_el).dato);
        last_el = (*last_el).el_succ;
        }
    }
```

Acquisizione di una grammatica

- Grammatica *G*: Max 10 produzioni
  - Vettore
- Produzione = parte sinistra + parte destra
  - Struttura
- Simbolo (terminale e non) = carattere
  - Parte sinistra = Parte destra = stringa

Acquisizione di una grammatica (I)

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

struct produzione {
   char sx[10]; /* parte sinistra */
   char dx[10]; /* parte destra */
};
typedef struct produzione TipoProd;
int acq_prod(TipoProd *);
```

Acquisizione di una grammatica (II)

```
int main() {
  TipoProd g[10]; int i, j, cont = 1;

for (i=0; i<10 && cont; i++) {
     printf("Inserire una produzione\n");
     cont = acq_prod(&g[i]);
     }

    for (j=0; j<i-1; j++)
       printf("%s->%s\n",g[j].sx,g[j].dx);
  }
```

Acquisizione di una grammatica (III)

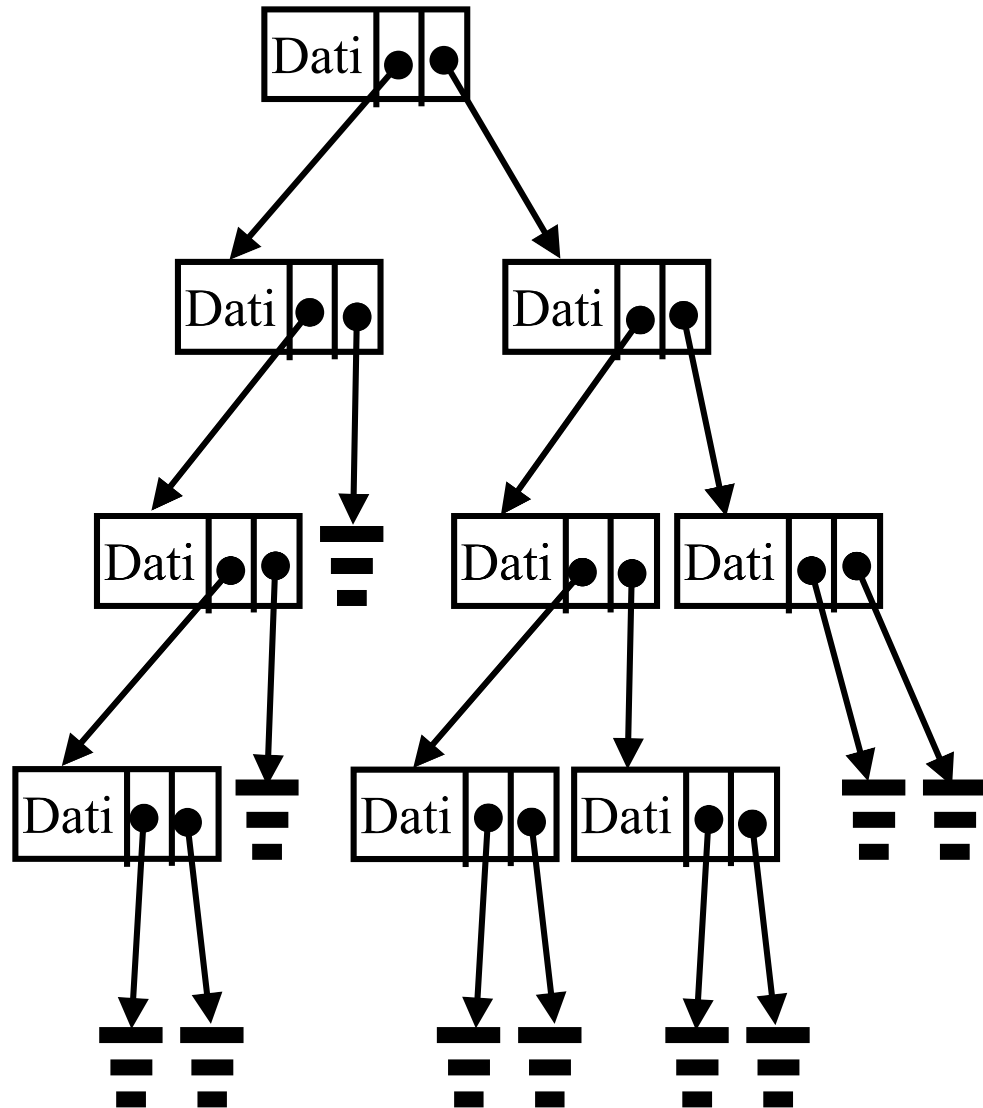
```
int acq_prod(TipoProd *produz) {
   printf("Parte sinistra ");
   printf("(* per terminare): ");
   scanf("%s", &produz->sx);
   if (produz->sx[0] != '*') {
      printf("Parte destra: ");
      scanf("%s", &produz->dx);
      }
   return (produz->sx[0] != '*');
   }
```

Esercizio

- Estendere il programma di acquisizione di una grammatica G
  - Usare una stringa per ciascun simbolo (terminale e non)
  - Prevedere una procedura di stampa della grammatica
- Eseguire un controllo minimale sugli inserimenti

Alberi binari

- Numero di figli fissato (2)
  - Accesso tramite attraversamento dell'albero
    - Ciascun elemento punta a 2 figli
    - Le foglie hanno un terminatore



Alberi binari

```
struct nodo_bin {
… /* dati */
struct nodo_bin *figlio_sx;
struct nodo_bin *figlio_dx;
} *albero_bin;

struct nodo_bin *new_el =
    (struct nodo_bin *)malloc(sizeof(struct nodo_bin));
elem->figlio_sx = new_el1;   elem->figlio_dx = new_el2;
```

Hash

Corso di Linguaggi di Programmazione

Università degli Studi di Bari

A.A. 2001/2002

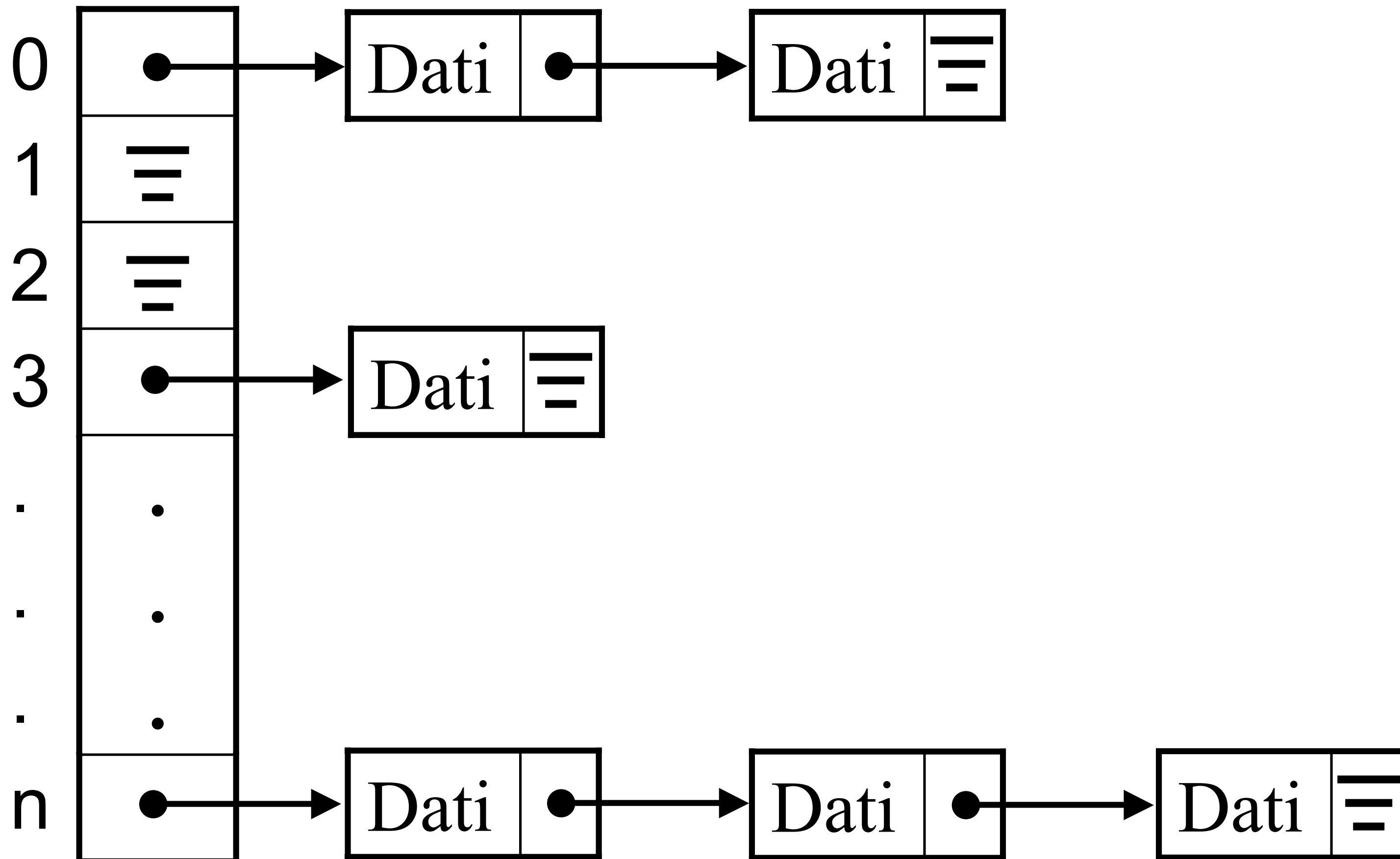
Ordinamento/Ricerca Hash

- Associazione a ciascun elemento di un valore univoco (*funzioni Hash*)
  - Calcolato sulla chiave
  - Usato come indice per la posizione nel vettore
- Elementi con uguale valore (*collisioni*)
  - Lista concatenata degli elementi con uguale valore (*sinonimi*)

Tabelle Hash

- Ogni elemento del vettore punta ad una lista di sinonimi
  - NULL se non esistono elementi con quel valore
- Ogni blocco della lista si riferisce a un elemento
  - Informazioni sull'elemento
  - Puntatore all'elemento successivo
    - NULL se non esiste

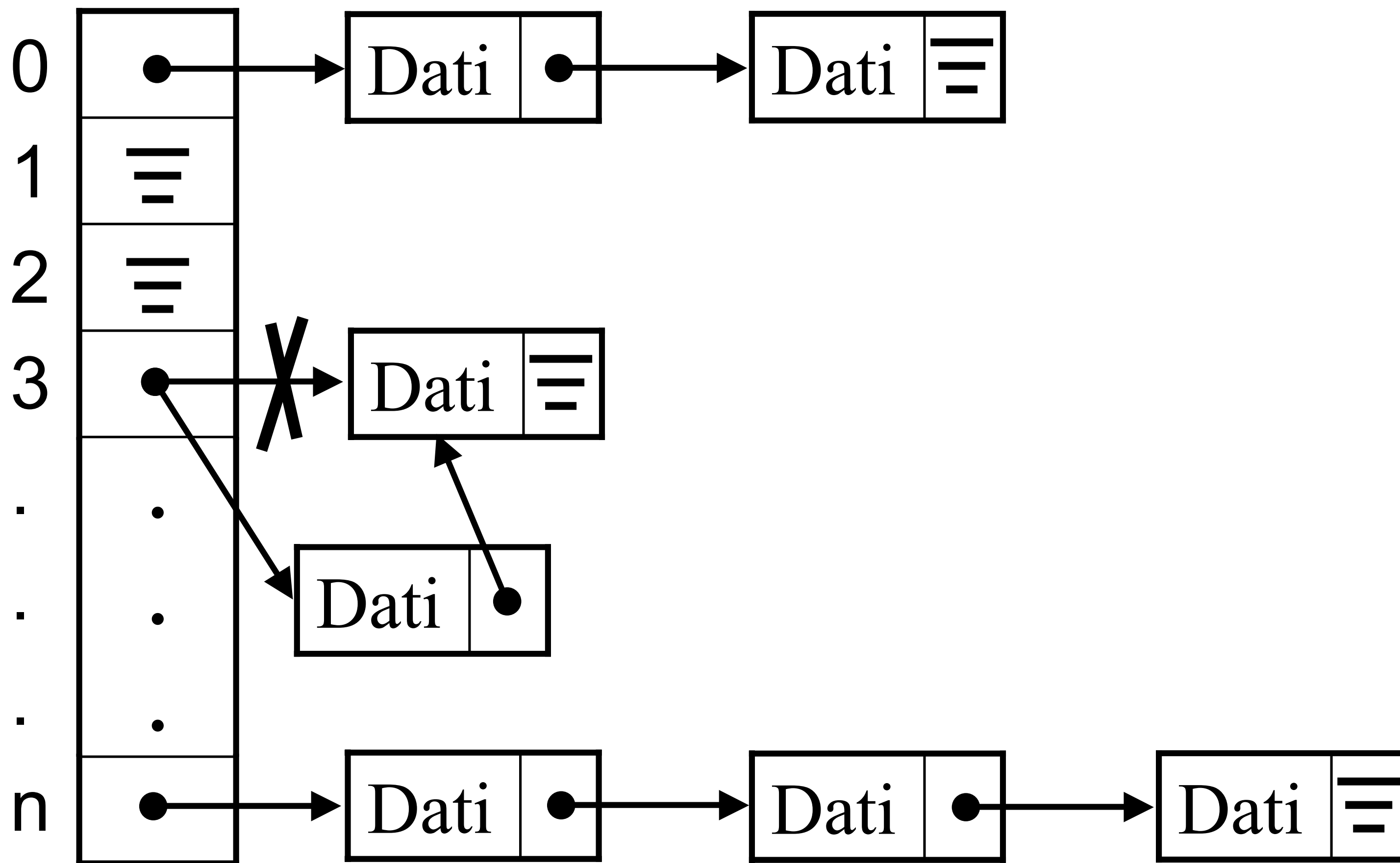
Schema di tabella Hash



Caratteristiche

- Tempo di ritrovamento costante
  – Poco più di un accesso in media
- Dimensionamento del vettore
  – Spazio sufficiente
  – Facilitazione della distribuzione
- Distribuzione dei valori
  – Funzioni con pochi sinonimi

Inserimento



Procedure Hash (I)

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

#define HASHSIZE 101

struct nlist {
   struct nlist  *next;
   char *name;
   char *defn;
   };

static struct nlist  *hashtab[HASHSIZE];
```

Procedure Hash(II)

```
unsigned hash(char *s) {
   unsigned hashval;
   for (hashval = 0; *s != '\0'; s++)
      hashval = *s + 31 * hashval;
   return hashval % HASHSIZE;
}

struct nlist  *lookup(char *s) {
   struct nlist  *np;
   for (np = hashtab[hash(s)]; np != NULL;
                           np = np->next)
      if (strcmp(s,np->name) == 0) return np;
   return NULL;
}
```

Procedure Hash(III)

```
char *strdup(char *s) {
   char *p;
   p = (char *)malloc(strlen(s) + 1);
   if (p != NULL) strcpy(p,s);
   return p;
}

struct nlist  *install(char *name, char *defn) {
   struct nlist  *np;
   unsigned hashval;
```

Procedure Hash(IV)

```
 if ((np = lookup(name)) == NULL) {
      np = (struct nlist  *)malloc(sizeof(*np));
      if (np == NULL ||
            (np->name = strdup(name)) == NULL)
         return NULL;
      hashval = hash(name);
      np->next = hashtab[hashval];
      hashtab[hashval] = np;
      }
   else
      free((void *)np->defn);
   if ((np->defn = strdup(defn)) == NULL)
      return NULL;
   return np;
}
```

Esercizio

- Completare il programma di gestione della tabella hash
  - Aggiungere il main
    - Lettura ciclica degli identificatori
  - Per ciascun elemento, invece della definizione, ricordare
    - numero d'ordine della sua prima occorrenza
    - numero d'ordine della sua ultima occorrenza

Esercizio (extra)

- Modificare il programma di gestione della tabella hash
  - Aggiungere gli elementi non trovati in coda alla lista

Problema

- Mantenere informazioni su un insieme di parole non noto a priori
  - Necessità di un ritrovamento efficiente
    - Ordinamento lessicografico
    - Ricerca binaria
  - Impossibilità di usare un vettore
    - Necessità di una struttura dinamica

Soluzione

- Albero binario ordinato lessicograficamente
  - Ordine mantenuto costantemente
  - Ricerca e Inserimento integrati
    - Si parte dalla radice, e se non è la parola cercata
      - Se la parola cercata è minore, esaminare il ramo sinistro
      - Se la parola cercata è maggiore, esaminare il ramo destro
    - Se, raggiunta la foglia, la parola non è stata trovata
      - Se la parola cercata è minore, inserirla nel ramo sinistro
      - Se la parola cercata è maggiore, inserirla nel ramo destro

Albero lessicale (I)

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

#define MAXWORD 100

struct tnode {
   char *word;
   struct tnode *left;
   struct tnode *right;
};
```

Albero lessicale (II)

```
int getword(char *, int);


struct tnode *addtree(struct tnode *,char *);

   struct tnode *talloc(void);
   char *strdup(char *);


void treeprint(struct tnode *);
```

Albero lessicale (III)

```
int main() {
   struct tnode *root;
   char word[MAXWORD];

   root = NULL;
   while (getword(word,MAXWORD) != 0)
         root = addtree(root,word);
   treeprint(root);

   system("PAUSE");
   return 0;
}
```

Albero lessicale (IV)

```
int getword(char *word, int lim) {
   printf("Inserisci una parola ");
   printf("(* per terminare): ");
   scanf("%s",word);
   word[lim] = '\0';
   return (word[0] != '*');
}

void treeprint(struct tnode *p) {
   if (p != NULL) {
      treeprint(p->left);
      printf("%s\n",p->word);
      treeprint(p->right);
      }
}
```

Albero lessicale (V)

```
struct tnode *addtree(struct tnode *p,char *w) {
   int cond;

   if (p == NULL) {
      p = talloc();
      p->word = strdup(w);
      p->left = p->right = NULL;
      }
   else if ((cond = strcmp(w,p->word)) == 0)
      printf("Gia' esistente!\n");
   else if (cond < 0)
      p->left = addtree(p->left,w);
   else
      p->right = addtree(p->right,w);
   return p;
}
```

Albero lessicale (VI)

```
struct tnode *talloc(void) {
   return
      (struct tnode *)malloc(sizeof(struct tnode));
}


char *strdup(char *s) {
   char *p;

   p = (char*)malloc(strlen(s) + 1);
   if (p != NULL)
      strcpy(p,s);
   return p;
}
```

Esercizio

- Completare il programma di gestione del dizionario (albero binario)
  – Contare il numero d’ordine progressivo di inserimento delle parole
  – Per ciascun elemento, memorizzare
    - numero d’ordine della sua prima occorrenza
    - numero d’ordine della sua ultima occorrenza

Alberi generici

- Numero di figli non fissato
  - Ciascun elemento punta ad una lista di figli
  - Ciascun figlio punta
    - all'elemento che rappresenta
    - al figlio successivo
  - Le foglie hanno un terminatore

Alberi generici

```
struct nodo_gen {
… /* dati */
struct figlio *primo_figlio;
} *albero_gen;

struct figlio {
struct nodo_gen *nodo;
struct figlio *figlio_succ;
};
```

Dati
Dati
Dati
Dati

Analisi Lessicale

Corso di Linguaggi di Programmazione

Università degli Studi di Bari

A.A. 2001/2002

File

var nome**: file of** *tipo*;	**FILE** ***puntatore;**
reset(nome**)**	puntatore **= fopen(**nome**, “r”);**
read(nome**,** arg**,** …**,** arg**)**	**fscanf(**puntatore**, “…”, &**arg**,** …**, &**arg);
rewrite(nome**)**	puntatore **= fopen(**nome**, “w”);**
write(nome**,** arg**,** …**,** arg**)**	**fprintf(**puntatore**, “…”,** arg**,** …**,** arg**);**
chiusura del file:	**fclose(**puntatore**);**
aggiunta di dati:	puntatore **= fopen(**nome**, “a”);**

File

- Gestione tramite un puntatore
  - Struttura contenente informazioni sul file
  - Modalità di apertura
  - Necessità di chiusura

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

- Possibilità di accesso per caratteri

```
int getc(FILE *puntatore)
int putc(int c, FILE *puntatore)
```

Estrazione di parole da file (I)

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

#define MAXTOKEN 256

struct token {
   char name[MAXTOKEN+1];
   int cat;
};

int gettoken(FILE *src, struct token *tkn, int max);
```

Estrazione di parole da file (II)

```
int main() {
   FILE *source;
   char *categories[] = {"3n caratteri",
            "3n+1 caratteri","3n+2 caratteri"};
   struct token next;

   source = fopen("sorgente","r");
   while (gettoken(source, &next, MAXTOKEN))
      printf("Token: %s -> Category: %s\n",
            next.name,categories[next.cat]);
   fclose(source);
   system("PAUSE");
   return 0;
}
```

Estrazione di parole da file (III)

```
int gettoken(FILE *src, struct token *tkn, int max) {
    int car, i=0;
    if ((car = getc(src)) != EOF) {
        while (isspace(car)) car = getc(src);
        while (car != EOF && !isspace(car) && i <= max) {
            tkn->name[i] = car;
            i++;
            car = getc(src);
        }
        tkn->name[i] = '\0';
        tkn->cat = i % 3;
        return 1;
    }
    else return 0;
}
```

Esercizio

- Inserire i token letti in una struttura ordinata
  - tabella Hash
  - albero binario
- Gestire token più lunghi del massimo consentito
  - Ignorare i caratteri in eccesso

Analizzatore lessicale

- Implementazione di un automa
  - Strutture a scelta multipla nidificate
- Output per ciascun token riconosciuto:
  - Classe
  - Contenuto (identificatori, interi, stringhe)
- Necessità di *lookahead*
  - Conservazione di caratteri letti ma non usati

Analizzatore lessicale
Un possibile insieme di token

- Operatori e Separatori

  / * + - > >= < <= <> = : := ; . , ( )

- Identificatori (lettera seguita da lettere o cifre)
- Caratteri, Interi, Reali
- Parole chiave

  BEGIN, BOOLEAN, CHAR, DO, ELSE, END, FALSE, IF, INTEGER, PROGRAM, REAL, THEN, TRUE, VAR

- Spazi bianchi, Fine file, Commenti **/*** … ***/**

Analizzatore lessicale
Possibili soluzioni

- Commenti
  - Ignorati dallo scanner
  - Ignorati dal main
- Identificatori – Keyword
  - Riconoscimento separato
  - Distinzione tramite procedura apposita

Analizzatore lessicale Funzioni

- `main();`
  - Legge iterativamente i token in un file, sistemando gli identificatori in una tabella Hash o albero binario
- `int scan();`
  - Legge il token successivo restituendone il codice ed eventualmente il contenuto (salta spazi e commenti)
- `int get_char();`
  - Legge il carattere successivo dal file sorgente, contando le linee (opzionale)
- `int keyword();`
  - Controlla se il token è una parola chiave, restituendone la classe

Analizzatore lessicale
Variabili

- `FILE *source_file`
  - File sorgente da analizzare
- `char id_or_no[]`
  - Contenuto dell’ultimo token letto (se identificatore o numero o stringa)
- `int lookahead`
  - Ricorda se è stato letto un carattere in più
- `int car`
  - Ultimo carattere letto

Analizzatore lessicale Implementazione (I)

```
if (!lookahead)
   car = get_char();
else
   lookahead = 0;
if (car == EOF)
   return FILE_END;
else if (isalpha(car))
   … /* riconoscimento identificatore */
else if (isdigit(car))
   … /* riconoscimento numero */
```

Analizzatore lessicale Implementazione (II)

```
switch (car) {
   case '*': return MULTIPLICATION;
   ...
   case '>':
      if ((car = get_char()) == '=')
         return GREATER_OR_EQUAL;
      else {
         lookahead = 1;
         return GREATER;
      }
   ...
   default: printf("Invalid char: %c\n",car);
}
```

Esercizio

- Scrivere un programma **C** che esegua l'analisi lessicale di un file e ne costruisca la tabella dei simboli (identificatori)
  - Assegnare un codice ad ogni token
  - Scrivere la funzione che legge e riconosce il token successivo
    - Implementazione di un automa

Analisi Sintattica

Corso di Linguaggi di Programmazione

Università degli Studi di Bari

A.A. 2001/2002

Realizzazione

- Grammatica non contestuale
  - no ricorsioni sinistre
  - LL(1)
    - produzione individuata dal primo token
- Ogni non-terminale -> funzione
  - rappresenta le produzioni che lo definiscono
  - non-terminali nella parte destra
    - invocazione ricorsiva di funzione
  - terminali nella parte destra
    - invocazione dello scanner

Espressioni

<condizione> ::= <espressione> (**<** | **<=** | **=** | **>=** | **>** | **<>**) <espressione> | **FALSE** | **TRUE**

<espressione> ::= <termine> {(**+** | **-**) <termine>}$_0$

<termine> ::= <fattore> {(* | /) <fattore>}$_0$

<fattore> ::= <intero> | <reale> | <identificatore> | **FALSE** | **TRUE** | <carattere> | **(** <espressione> **)** | (**+** | **-**) <fattore>

<identificatore>, <intero>, <reale>, <carattere>: vd. scanner

Istruzioni

<blocco> ::= **BEGIN** <lista di istruzioni> **END**

<lista di istruzioni> ::= <istruzione> {**;** <istruzione>}$_0$

<istruzione> ::= <assegnazione> | <istruzione while> | <blocco>

<assegnazione> ::= <identificatore> **:=** <espressione>

<istruzione if> ::= **IF** <condizione> **THEN** <istruzione> [**ELSE** <istruzione>]

<istruzione while> ::= **WHILE** <condizione> **DO** <istruzione>

Programmi

<programma> ::= **PROGRAM** <identificatore> **;**
<corpo> **.**

<corpo> ::= [<sezione dichiarazioni>] <blocco>

<sezione dichiarazioni> ::= **VAR** {<lista di identificatori>
: <tipo> **;**}$_0$

<tipo> ::= **CHAR** | **BOOLEAN** | **REAL** | **INTEGER**

Assunzioni

- Ciascuna funzione trova il primo token già letto in memoria
- Semplice segnalazione degli errori

Simboli notevoli

- Starter
  - terminale che può iniziare l'espansione di un non terminale
- Follower
  - terminale che può seguire l'ultimo dell'espansione di un non terminale
- Stopper
  - terminale che non può apparire nell'espansione di un non terminale

Esempio: <fattore>

- Starters:

  `INT,REAL,ID,TRUE,FALSE,OPEN_PAR,ADD,SUB`

- Followers:

  `DIV,MUL,ADD,SUB,GREATER,GREATER_EQ,LESS,`
  `LESS_EQ,EQ,NOT_EQ,CLOSE_PAR,SEP,DO,END`

- Stoppers:
  - (Esercizio)

Esempio: <fattore>

```
int fattore(…) {
   switch (next.cat) {
      case ID:TRUE:FALSE: gettoken(…);
         break;
      case OPEN_PAR: gettoken(…); epressione(…);
         if (next.cat == CLOSE_PAR) gettoken(…)
           else error;
         break;
      case ADD:SUB: gettoken(…); fattore(…);
         break;
      default: error;
   }
}
```

Esempio: <termine>

```
int termine(…) {
   switch (next.cat) {
      case ID:ADD:SUB:OPEN_PAR:TRUE:FALSE:
         fattore(…);
         while (next.cat == MUL || next.cat ==
  DIV) {
            gettoken(…);
            fattore(…);
         }
      default: error;
   }
}
```

Esempio: <espressione>

```
int espressione(…) {
    switch (next.cat) {
        case ID:ADD:SUB:OPEN_PAR:
            termine(…);
            while (next.cat == ADD || next.cat ==
   SUB) {
                gettoken(…);
                termine(…);
            }
        default: error;
    }
}
```

Esercizio

- Completare le procedure presentate per l'analisi sintattica di espressioni
  - Gestire i valori diretti oltre agli identificatori
    - `INT, REAL, CHAR`
- Completare il parser per l'analisi sintattica del linguaggio presentato
- Extra:
  - Gestire gli errori
    - Riallineamento del parsing ai token attesi