Università degli Studi di Verona
A.A. 2017-2018

APPUNTI DI PROGRAMMAZIONE 1

Creato da: Davide Zampieri

PRIMO SEMESTRE

INCLUSIONE LIBRERIE

#include <nome_libreria.h>

Librerie utilizzabili:

- stdio.h --> per le funzioni printf e scanf
- stdlib.h --> per le funzioni rand e srand
- time.h --> per la funzione time
- stdbool.h --> per usare variabili di tipo bool con valori true/false
- math.h --> per le funzioni abs, fabs, pow

COMPILAZIONE

gcc -Wall -lm prog.c -o prog

- -Wall --> mostra tutti i warning
- -lm --> include la libreria math
- -o --> crea un file eseguibile

FUNZIONE SCANF

scanf("%i", &var);

- %i --> tipo della variabile
- &var --> il valore letto viene assegnato alla variabile var

VARIABILE

- dichiarazione: *int var;*
- assegnamento: *var = 0;*

TIPI

- numeri interi: *int*
- numeri in virgola mobile (32 bit): *float*
- numeri in virgola mobile (64 bit): *double*
- carattere: *char*

La funzione *sizeof(int)* restituisce la dimensione in byte (di tipo long unsigned) di un determinato tipo o del tipo di una determinata variabile.

INT

- %i --> in generale
- %d --> in base 10
- %o %#o --> in base 8 (per inserire un numero in base 8 anteporre 0)
- %x %#x %X %#X --> in base 16 (per inserire un numero in base 16 anteporre 0x)

FLOAT E DOUBLE

- %f --> formattazione normale (float)
- %lf --> formattazione normale (double)
- %e --> notazione scientifica
- %g --> notazione migliore

CARATTERE

- %c --> formattazione
- attenzione a mettere " %c" nella scanf per ignorare spazi e invii

SPECIFICATORI

spec tipo nome;

- unsigned (int) --> %u
- short (int) --> %hi
- long (int e double) --> %li %Lf
- long unsigned (int) --> %lu

CONVERSIONE

- int --> double/float: non si perde nulla
- double/float --> int: si perde la parte decimale
- cast --> int a = 3; int b = 2; double c = a/b (= 1.0); double c = (double) a/b (=1.5)

ESPRESSIONI DI CONFRONTO

<
>
<=
>=
==
!=

!	not
&&	and
||	or

OPERATORI DI ASSEGNAMENTO

a = a + 1

a += 1 (esistono anche -=, *=, /=, %=)

Forma prefissa: il valore dell'espressione è quello della variabile dopo l'incremento

- ++a (esiste anche --a)

Forma postfissa: il valore dell'espressione è quello della variabile prima dell'incremento

- a++ (esiste anche a--)

FORMATTAZIONE PRINTF

- %2i --> il numero occupa due spazi
- %.2f --> il numero ha due cifre dopo la virgola

CICLO FOR

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

CICLO WHILE / DO-WHILE

```
int i=1;
while(i<=n){
    ...
    i++;
}

do {
    ...
} while(...);
```

IF ELSE

```
if (espressione_1) {
    istruzione_1;
} else if (espressione_2) {
    istruzione_2;
} ... {
    ...
} else {
    istruzione_3;
}
```

CONDIZIONALE

condizione ? istruzione_1 : istruzione_2;

Se la condizione è vera esegue l'istruzione_1, altrimenti l'istruzione_2.

SWITCH

```
switch (espressione) {
    case valore_1:
        istruzione_1;
        ...
        break;
    ...
    default:
        istruzione_d;
        ...
        break;
}
```

Se non si mette il break, quando trova vero esegue tutti i casi successivi.

BREAK / CONTINUE

- break --> permette di uscire immediatamente dal ciclo che si sta eseguendo. Utilizzata per terminare un ciclo perché si è verificata una certa condizione.
- continue --> determina l'uscita dall'iterazione corrente e si prosegue con l'esecuzione di una nuova iterazione.

NUMERI CASUALI

- srand(time(NULL)) --> seme iniziale; se si mantiene il seme di default, ogni volta verrà generata la stessa sequenza
- n = rand() % 2 --> valore casuale tra 0 e 1
- n = rand() % 100 --> valore casuale tra 0 e 99

- n = rand() % 100 + 1 --> valore casuale tra 1 e 100
- n = rand() % 100 + 23 --> valore casuale tra 23 e 123
- n = (rand() % (max - min + 1)) + min --> valore casuale tra min e max

DICHIARAZIONE ARRAY

int vettore[10]; --> array di interi di dimensione massima 10

INIZIALIZZAZIONE ARRAY

diretta --> int vettore[5] = {1,2,3,4,5};
primo elemento 1, gli altri a 0 --> int vettore[5] = {1};
tutti gli elementi a 0 --> int vettore[5] = {0};
assegnamento dei valori agli elementi dell'array -->

```
vettore[0] = 10;
...
vettore[9] = 2;
```

MATRICE

A righe e B colonne --> int matrice[A][B];
matrice 4x3 --> int matrice[4][3];
terza riga seconda colonna --> matrice[2][1] = 5;
gli elementi vengono specificati per riga --> int matrice[2][4] = { {1,2,3,4}, {5,6,7,8} };

MACRO

#define NOME_costante valore_costante
Non viene allocata in memoria, ma sostituisce il valore quando incontra il nome.

CONST

const char vettore[DIM] = {...};
La variabile dichiarata dopo non può essere modificata.

FUNZIONI

dichiarazione --> *int sum(int, int);*
definizione -->

```
int sum(int a, int b) {
   return a + b;
}
```

funzione che non ha valore di ritorno e non ha argomenti --> *void funzione(void) { ... }*

VARIABILI LOCALI (AUTOMATICHE) E STATICHE

```
void funzione(void) {
   int n = 1;              //variabile automatica: viene ricreata ogni volta
   static int nS = 1;      // variabile statica: viene creata solo una volta
                           // NB: l'inizializzazione avviene solo al momento della creazione
                           // NB: quando non specificato, sono inizializzate a zero
}
```

FUNZIONI E VETTORI

- le funzioni non possono ritornare vettori (perché i vettori passati vengono modificati direttamente)
- passare un vettore ad una funzione --> void funzione(int vettore[], int lunghezzaVettore);
- passare una matrice ad una funzione -->
  void funzione(int matrice[10][3]);
  void funzione(int matrice[][3]);

SECONDO SEMESTRE

VARIABILI LOCALI E GLOBALI

- *Locali*: definite all'interno di una funzione (possono essere utilizzate solo all'interno della funzione)
- *Globali*: definite all'esterno di qualsiasi funzione (possono essere utilizzate da qualsiasi funzione)
- Valore di default: di una *variabile globale* è zero; le *variabili locali* devono essere inizializzate

VARIABILI AUTOMATICHE E STATICHE

- *Automatiche*: dichiarate all'interno di una funzione e create ogni volta che la funzione viene creata
- *Statiche*: quando la funzione termina, mantengono il valore per la prossima invocazione della funzione

NUMERI CASUALI

- srand(time(NULL)) --> seme iniziale per non generare la stessa sequenza ogni volta
- n = rand() % 100 --> valore casuale tra 0 e 99
- n = rand() % 100 + 1 --> valore casuale tra 1 e 100
- n = rand() % 100 + 23 --> valore casuale tra 23 e 123
- n = (rand() % (max - min + 1)) + min --> valore casuale tra min e max

FUNZIONI RICORSIVE

- Una funzione è detta *ricorsiva* se include una chiamata a sé stessa:

```
int factorial(int n) {
    if(n == 0)        // passo base
        return 0;
    else              // passo ricorsivo
        return n * factorial(n – 1);
}
```

STRUTTURE

La *struttura* permette di raggruppare elementi di diverso tipo in un'unica entità logica e definisce un nuovo tipo di dato: si possono dichiarare variabili di tipo *struct nome_struttura*

Definizione:	Dichiarazione e accesso ai campi:
`struct date {` `  int day;` `  int month;` `  int year;` `};`	`struct date today;` `today.day = 10;` `today.month = 3;` `today.year = 2014;`

Inizializzazione:	Assegnamento composto:
`struct date d...` `... = {10, 3, 2014};` `... = {10, 3}; // d.year non iniz.` `... = {.month = 3, .year = 2014};`	`struct date today;` `...` `today = (struct date) {10,3,2014};`

Le strutture possono essere utilizzate come argomenti di funzioni, ma qualsiasi modifica apportata dalla funzione non ha effetto sulla struttura originale bensì solo sulla copia che viene creata alla chiamata di funzione

ARRAY DI STRUTTURE

- Gli elementi di un array di strutture hanno come tipo una struttura: *struct date appointments[10];*
- Accesso ad una struttura all'interno dell'array: *struct date d = appointments[3];*

STRUTTURE CONTENENTI STRUTTURE

- È possibile definire una struttura che contiene altre strutture come suoi membri:
```
struct dateAndTime {
    struct date d;
    struct time t;
};
```
- Accesso ad un membro di una struttura membro: `event.d.day = 17;`
- Inizializzazione con un'unica istruzione di una struttura di strutture:
```
struct dateAndTime event = { {12, 3, 2013}, {14, 30} };
```

STRUTTURE CONTENENTI ARRAY

- È possibile definire una struttura che contiene array come suoi membri:
```
struct month {
    int numberOfDays;
    char name[3];
};
```
- Inizializzazione con un'unica istruzione: `struct month m = { 31, {'M', 'A', 'R'} };`

PROGRAMMI COMPOSTI DA PIU' FILE

- *File header*: file con estensione .h in cui vanno incluse le dichiarazioni delle funzioni del programma
- *Inclusione*: nel file del programma principale si scrive #include "funzioni.h"
- *Compilazione*: si usa il comando gcc *.c (es. funzioni.h, funzioni.c, main.c)
- *Dipendenze*: per evitare di copiare due volte il contenuto dei file header scrivere il seguente codice
```
#ifndef FUNZIONI_H
#define FUNZIONI_H
...
#endif
```

STRINGHE DI CARATTERI

- Array di caratteri: *char array_car[] = {'H', 'e', 'l', 'l', 'o'}*
- Per le stringhe si utilizza il carattere '\0' per segnalarne la fine: *char stringa[] = {'H', 'e', 'l', 'l', 'o', '\0'}*
- Inizializzazione di una stringa (di dimensione 6): *char word[] = {"Hello"}* / *char word[] = "Hello"*
- Accesso ai singoli caratteri di una stringa (con l'usuale notazione degli array): *word[2]* → 'l'
- Stampa di un array di caratteri che termina con il carattere nullo: *printf("%s\n", word)*

- Lettura di una stringa di caratteri (aggiunge automaticamente il carattere nullo alla fine):
```
char word[81];
scanf("%80s", word);
```
- Legge finché non incontra uno spazio, un carattere di tabulazione o la fine linea: se l'input è *"Hello world"* viene letto e assegnato a *word* solo *"Hello"*

PUNTATORI

- Dichiarazione, creazione e accesso indiretto:
```
int x = 3;              // variabile intera
int *pointer;           // puntatore ad int (può contenere la posizione in memoria di un int)

pointer = &x;           // assegnamento di un puntatore a x (operatore di indirizzamento &)
int y = *pointer;       // assegna ad y ciò a cui punta pointer (operatore di indirezione *)
*pointer = 5;           // assegna alla variabile puntata da pointer (cioè x) il valore 5
printf("%i", x);            // → 5
printf("%i", *pointer);     // → 5
printf("%i", y);            // → 3
```
- Precedenze: gli *operatori di indirizzamento & e di indirezione* * hanno precedenza più alta rispetto a tutti gli *operatori binari* del C; l'*operatore .* di accesso ai membri della struttura ha precedenza più alta rispetto agli *operatori di indirezione*
- Attenzione: non posso restituire un puntatore ad una variabile dichiarata in una funzione perché quando la funzione termina la sua zona sullo stack viene cancellata

PUNTATORI A STRUTTURE

```
struct date *date_ptr;
struct date struct_date = {23, 12, 2017};
date_ptr = &struct_date;
```

`int day = (*date_ptr).day;` `(*date_ptr).month = 5;`	`int day = date_ptr->day;` `date_ptr->month = 5;`

Operatore dei puntatori a struttura ->: *(*x).y* può essere riscritto come *x->y*

PUNTATORI E FUNZIONI

- Il valore del puntatore viene copiato nel parametro formale quando la funzione viene chiamata
- Qualsiasi modifica al valore del puntatore non ha effetto sul puntatore che è stato passato
- Al contrario, i dati ai quali il puntatore fa riferimento possono essere modificati (side-effect)

ALLOCAZIONE DINAMICA DELLA MEMORIA

- *Allocazione statica:* le variabili vengono allocate automaticamente in memoria (sullo stack) quando si entra in un blocco di codice e corrispondentemente vengono distrutte automaticamente quando si esce dal blocco stesso; la dimensione degli array deve essere definita con costanti
- *Allocazione dinamica*: consente di determinare lo spazio necessario a certe variabili durante l'esecuzione del programma; una variabile viene allocata dinamicamente in memoria (sullo heap) attraverso specifiche istruzioni, e rimane tale finché non viene esplicitamente deallocata

- *Funzione malloc()*: consente di allocare dinamicamente la memoria; richiede il numero totale di byte da allocare in memoria e restituisce un puntatore all'area di memoria allocata, che vale NULL se l'allocazione non è stata possibile (es. quando si chiede più memoria di quella disponibile)
- *Operatore sizeof()*: è usato per determinare la dimensione degli elementi da riservare con la malloc()
- *Funzione free()*: permette di liberare la memoria allocata dinamicamente dalla malloc(); se uso il puntatore alla zona di memoria appena liberata ho un dangling pointer (errore); dopo la free() lo spazio di memoria non si cancella ma rimane libero per la prossima malloc(); se non libero la memoria dinamica ho un memory leak (spreco di spazio)

```
int *int_ptr;
...
int_ptr = (int *) malloc(sizeof(int));        // attenzione al cast
if (int_ptr == NULL) {                        // verifica che l'operazione sia andata a buon fine
    ...
}
...
free(int_ptr);
```

PUNTATORI E ARRAY

- Nella dichiarazione di un puntatore ad array si specifica solo il tipo degli elementi che formano l'array, perciò può essere inizializzato come un puntatore ad un intero (quindi il puntatore ad un array è in realtà un puntatore al primo elemento contenuto nell'array)

```
int *array_ptr;                   // puntatore ad un array di interi
int array[10];                    // array di interi
array_ptr = array;
array_ptr = &array[0];            // equivale a sopra
*array_ptr = 1;                   // equivale ad array[0]
*(array_ptr + 3) = 4;             // equivale ad array[3]
```

- Quando si passa un array ad una funzione in realtà viene passato un puntatore all'array e quindi è possibile modificare in modo "permanente" l'array passato alla funzione
- Per creare una matrice basta dichiarare un array di puntatori (quindi un puntatore a un puntatore)

```
char **stringhe = (char **) malloc(sizeof(char *) * N);
stringhe[i] = (char *) malloc(sizeof(char) * N);
stringhe[i] = ...;

float **M = (float **) malloc(sizeof(float *) * N);
M[i] = (float *) malloc(sizeof(float) * N);
M[i][j] = ...;
```

FUNZIONI SULLE STRINGHE

int strcmp(const char *s1, const char *s2); *Confronta la stringa s1 con s2, ritorna 0 se le due stringhe sono uguali, -1 se s1 < s2, oppure 1 altrimenti (il segno del risultato corrisponde al risultato della differenza tra s1 e s2).*	int my_strcmp(const char *s1, const char *s2) { int pos = 0; while(s1[pos] == s2[pos] && s1[pos] != '\0') pos++; return s1[pos] - s2[pos]; }

char *strcpy(char *s1, const char *s2); *Copia la stringa s2 nella stringa s1, incluso il carattere di terminazione '\0'.*	char *my_strcpy(char *s1, const char *s2) { int pos = 0; do { s1[pos] = s2[pos]; } while(s2[pos++] != '\0'); return s1; }
size_t strlen(const char *s); *Restituisce la lunghezza della stringa s.*	int my_strlen(const char *s) { int len = 0; while(*s++ != '\0') len++; return len; }
Funzione per la lettura di una riga di caratteri come unica stringa.	char *read_line(void){ char s[257]; char c; int pos = 0; do { c = getchar(); if (c != '\n') s[pos++] = c; else s[pos++] = '\0'; } while (c != '\n'); return s; }
Funzione ricorsiva sugli array.	void somma(double *S, double *C, double *R, int length) { if(length > 0) { *S = *C + *R; somma(S+1, C+1, R+1, length-1); } }

FUNZIONI SULLE LISTE

Struttura che rappresenta un nodo di una lista di type.	struct node_t { type head; struct node_t *tail; };
Funzione che crea un nuovo nodo.	struct node_t *crea(type val, struct node_t *next){ struct node_t *n = malloc(sizeof(struct node_t)); n->head = val; n->tail = next; return n; }
Funzione con current.	void function(struct node_t *n) { struct node_t *current = n; if(current == …) { … } else { while(current != NULL) { if(current->next == …) { … break; } current = current->next; } } }

Funzione con current e prev.	void function(struct node_t *n) { struct node_t *current = n; struct node_t *prev = NULL; if(current == …) { … } while(current != NULL) { if(current == …) { … } prev = current; current = current->next; } }
Funzione ricorsiva che crea una lista partendo da una stringa.	struct node_t *crea_lista(char *s) { if(…) // caso limite return crea_lista(s+1); struct node_t *n = malloc(sizeof(struct node_t)); n->head = *s; if(*(s+1) == '\0') // passo base n->tail = NULL; else // passo ricorsivo n->tail = crea_lista(s+1); return n; }
Funzione per la stampa di una lista.	void stampa_lista(struct node_t *nodo) { if(nodo->tail == NULL) { printf("%c\n", nodo->head); } else { printf("%c-", nodo->head); stampa_lista(nodo->tail); } }
Funzione che libera la memoria dinamica occupata da una lista.	void svuota_lista(struct node_t *n) { struct contatto_t *current = n; struct contatto_t *prev = NULL; while(current != NULL) { if(prev != NULL) free(prev); prev = current; current = current->next; } if(prev != NULL) free(prev); }