

Programmazione 1

Corso di Laurea in Informatica - Università degli studi di Verona

FEDERICO BRUTTI

Federico Brutti
`federico.brutti@studenti.univr.it`

Devi sbatterce 'a capoccia - Davide B.

Indice

5 | Introduzione

| | | |
|-----|--|---|
| 1.1 | Definizione di algoritmo e linguaggi di programmazione | 5 |
| 1.2 | Condizioni e Cicli | 5 |

6 | Il linguaggio C - basi

| | | |
|-----|--|---|
| 2.1 | Introduzione a C | 6 |
| 2.2 | I/O, variabili e conversioni | 6 |
| 2.3 | Costrutto condizionale, scrittura di cicli | 6 |
| 2.4 | Vettori mono- e bidimensionali | 6 |

7 | Strutture e Funzioni

| | | |
|-----|---|---|
| 3.1 | Strutture dati e vettori di strutture | 7 |
| 3.2 | Sottoprogrammi | 7 |

8 | Stringhe e Puntatori

9 | Memoria Dinamica e Liste

| | | |
|-----|------------------------|----|
| 5.1 | Memoria Dinamica | 9 |
| 5.2 | Liste Dinamiche | 10 |
| 5.3 | Esercizi | 14 |

| Ulteriori Librerie, Funzioni e QOL

| | | |
|-----|---|----|
| 6.1 | Libreria ctype.h | 16 |
| 6.2 | Switch(), case, break | 16 |
| 6.3 | Argc, Argv | 16 |
| 6.4 | Divisione di un progetto su più files | 16 |
| 6.5 | Debugger GDB | 17 |

18 | Soluzioni degli Esercizi

| | | |
|-----|--------------------------------|----|
| 7.1 | Esercizi di base | 18 |
| 7.2 | Condizioni e cicli | 18 |
| 7.3 | Vettori e Matrici | 18 |
| 7.4 | Funzioni e Struct | 18 |
| 7.5 | Stringhe e Puntatori | 19 |
| 7.6 | Memoria dinamica e Liste | 19 |

Introduzione

- 1.1 Definizione di algoritmo e linguaggi di programmazione
- 1.2 Condizioni e Cicli

Il linguaggio C - basi

2.1 Introduzione a C

2.2 I/O, variabili e conversioni

2.3 Costrutto condizionale, scrittura di cicli

2.4 Vettori mono- e bidimensionali

Strutture e Funzioni

3.1 Strutture dati e vettori di strutture

3.2 Sottoprogrammi

Stringhe e Puntatori

Memoria Dinamica e Liste

5.1 Memoria Dinamica

Finora hai potuto vedere il funzionamento della memoria statica; la quale viene sempre allocata mediante il compilatore prendendo una zona riservata della stack prima che il programma venga attivato e non sarà possibile modificarne i limiti.

L'utilizzo della memoria dinamica porta la soluzione a questo problema, poiché consente di allocare memoria durante l'esecuzione del programma. Per poterlo fare è necessario introdurre la libreria digitale **stdlib.h**, che porta con sé le seguenti funzioni:

- *malloc(dimensione)*; Consente di allocare un determinato numero di bytes nella heap.
- *calloc(totale, dimensione)*; Alloca per un determinato numero di elementi un determinato numero di bytes nella heap.
- *realloc(puntatore, dimensione)*; Utilizzata se vi è la necessità di aggiungere ulteriore memoria mantenendo i dati già salvati¹.
- *free(puntatore)*; Consente di liberare lo spazio precedentemente utilizzato.

La modalità preferibile per allocare spazio è utilizzare l'operatore **sizeof(variable)**, che ricava automaticamente la dimensione in byte di una determinata variabile². Tuttavia, attenzione, perché c'è differenza fra le seguenti scritture:

```
int *ptr;
...
malloc(sizeof(ptr));           //Alloca la dimensione del puntatore.
malloc(sizeof(*ptr));          /*Alloca la dimensione del dato all'indirizzo
...                             del puntatore.*/
```

La memoria dinamica si comporta *come un array* e il tipo di dato è specificato dal puntatore. È possibile accedervi utilizzando un indice come è fattibile la dereferenziazione per accedere al primo elemento.

Ricorda sempre di *liberare la memoria* quando hai finito di utilizzarla; non vorresti mai degli spazi ancora occupati per un programma la cui esecuzione è già terminata, nevvvero?

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

```
int main(){
```

¹Se non riesce ad allocare memoria, ritorna un puntatore a NULL. Puoi usare questo comportamento a tuo vantaggio per effettuare dei controlli.

²Non riesce, tuttavia, a calcolare la memoria dinamica e quindi va indicato il numero di variabili per cui allocare la memoria.

```

//Dichiaro la variabile ed alloco la memoria
int *ptr;
ptr = calloc(5, sizeof(*ptr));

//Inserisco i numeri in posizioni precise
*ptr = 1;
ptr[1] = 2;
ptr[2] = 3;
ptr[4] = 5;

//Stampo quanto dato
printf("%d, %d, %d, %d, %d\n", ptr[0], ptr[1], ptr[2], ptr[3], ptr[4]);

//Libero la memoria
free(ptr);
ptr = NULL;

return 0;
}

```

Il meme più conosciuto di C è forse quello dei **Memory Leaks**. Questo problema avviene quando la memoria allocata non viene mai liberata e le conseguenze riguardano il rallentamento del computer. Tre consigli per evitare che ciò succeda:

- Controlla gli errori mediante costrutti condizionali, sicché non vi siano errori quando la memoria non può essere usata.
- Libera SEMPRE la memoria quando smetti di utilizzarla.
- Setta il puntatore a NULL dopo aver liberato la memoria per essere sicuro che non si comporti in modo strano.

Quanto visto vale per le singole variabili e va benissimo; normalmente vengono chiesti però gli **Array Dinamici** con tutte le loro relative funzioni che sono già state trattate in precedenza. La loro dichiarazione segue la seguente sintassi e può essere utilizzata con ogni tipo di dato; anche se user-defined. Useremo come esempio int:

```

//Dichiaro il puntatore
int *ptr;

//Cast forzato a puntatore variabile e alloca la dimensione del tipo.
ptr = (int*) malloc(sizeof(int));

```

5.2 Liste Dinamiche

Parliamo ora dell'ultimo argomento del corso, obiettivamente il più difficile da immaginare e da rispecchiare sul codice: le **Liste Dinamiche**. Si tratta di un costrutto di tipo ricorsivo che permette di legare come *nodi* degli insiemi di dati.

Personalmente mi ha aiutato molto immaginare le liste come uno di quei magneti delle uova di Pasqua, perché come vedrai, ci è possibile aggiungere un elemento in testa o in coda ad essa, modificare il nodo desiderato, distruggere tutto e, ovviamente, visualizzare il contenuto della lista. Non indugiamo oltre e studiamo i metodi per lavorare con questo costrutto.

Una lista dinamica necessita prima di tutto due *tipi strutturato*; uno che contenga l'intera lista ed uno sotto forma di puntatore per spostarsi fra i nodi. Tutti gli altri elementi dipenderanno da ciò che si vuole creare; per esempio nella creazione di un registro scolastico verranno usate delle stringhe al posto dei nostri due interi. Dichiarata la struttura, è buona prassi far *puntare la lista a NULL* per evitare errori.

```
typedef struct node{           //Dichiarazione dello strutturato "node"
    int a;
    int b;
    struct node *next;         //puntatore al nodo seguente
}lista;                        //Dichiarazione della lista "lista"

lista* l1 = NULL;              //Puntare a nulla per poter lavorare
```

Dichiarato ciò, è ora possibile lavorare sulla lista. Vediamo una ad una tutte le funzioni base che ti serviranno. Ricorda che qualunque funzione debba modificare la lista, avrà come tipo di dato una variabile puntatore del costrutto. Inoltre, un piccolo monito: non sperare nemmeno di passare l'esame senza conoscere questo argomento.

- Inserimento in testa:

Siccome dobbiamo modificare la struttura della lista, la funzione è un puntatore a lista. Per far sì che funzioni è necessario dichiarare un'altra variabile temporanea *tmp* sulla quale avverrà l'effettivo lavoro, per evitare di perdere i dati sui quali si andrà a lavorare.

Ci sono due metodi per inserire in testa: passare le variabili alla funzione oppure richiederle nella stessa. Ritengo la seconda opzione la migliore in quanto permette una scrittura più pulita ed organizzata, ma ora entriamo nel vivo.

Nel caso in cui *tmp* punti a qualcosa, sarà possibile inserire dati. Qui si richiede l'input da tastiera (o si assegnano le variabili allo spazio apposito) per poi assegnare la lista iniziale al nodo seguente di *tmp*. Abbiamo ora inserito *in testa a "lista" l'insieme di tmp*. Infine si assegnerà alla lista la variabile lavorata *tmp* per salvare quanto fatto. Ritornerà la variabile lista.

```
lista *insTesta(lista* l1){

    lista *tmp;                //Per tenere la testa puntata
    tmp = (elem*) malloc(sizeof(elem)); //Creo lo spazio per tmp

    if(tmp != NULL){
        scanf("%d%d", &tmp->a, &tmp->b);
        tmp->next = l1;         //La testa segue il nuovo nodo
        l1 = tmp;               //Aggiorno la lista
    } else {
        printf("Memoria esaurita!\n"); //Un semplice failproof
    }
}
```

```

    return l1;                                //Ritorno la lista modificata
}

```

- Inserimento in coda:

L'inserimento in coda modifica la struttura principale della lista solo quando questa è vuota. In caso contrario la funzione sarà *void*. Si richiede la dichiarazione di un'altra variabile puntatore *prev*, insieme a *tmp*, con lo scopo di poterle assegnare il valore della testa e scorrerla per darle i valori di *tmp* finché non finisce.

Se *tmp* punta a qualcosa, è possibile inserire dati in essa e in tal caso, se la lista principale *l1* è finita, ovvero punta a *NULL*, le si dà tutto il contenuto di *tmp*. In caso contrario, si scorre l'intera lista assegnando *l1* a *prev*, fintanto che non finisce, assegna il valore di *tmp* al *prossimo nodo di prev* e passa al prossimo nodo.

```

lista *insCoda(lista *l1){

    lista *prev;
    lista *tmp;
    tmp = (elem*) malloc(sizeof(elem));

    if(tmp != NULL){
        tmp->next = NULL;                //Il prossimo nodo punterà al nulla
        scanf("%d%d", &tmp->a, &tmp->b);

        if(l1 == NULL){                  //Se l1 punta al nulla
            l1 = tmp;                    //l1 diventa tmp
        } else {                          //Altrimenti scorri l'intera lista
            for(prev = l1; prev->next != NULL; prev = prev->next){
            }
            prev->next = tmp;
        }
    } else {
        printf("Memoria esaurita!\n");
    }

    return l1;
}

```

- Ricerca di un elemento specifico:

Classica richiesta base da esame; per poterla realizzare avrai bisogno di una funzione che riceva la lista e l'elemento da cercare³ per poi riportarne l'eventuale presenza o assenza.

Crea la tua solita variabile *tmp* e scorrla interamente spostandoti dallo stato corrente a quello successivo; aggiungi una condizione per verificare la presenza del ricercato ed infine ritorna il risultato.

```

int findEl(lista *l1, int n){

    int flag;
    lista *tmp;

```

³Se preferibile, puoi richiederlo all'interno della funzione.

```

tmp = (lista*) malloc(sizeof(lista));
tmp = l1;

for(; tmp; tmp = tmp->next){}          //Scorri la lista
    if(tmp->a == n){
        flag = 1;
    }
}

return (flag);
}

```

- Eliminazione di un elemento della lista:

La rimozione di un elemento richiede la dichiarazione di tre variabili puntatori lista in una funzione che riceve la lista primaria e l'elemento killer che rimuoverà il nodo.

Scorri la lista; se l'elemento è trovato salvalo in una variabile *canc*, che verrà liberata in seguito, e lega il nodo successivo a quello precedente, così da non lasciare spazi vuoti nella lista. Ritorna infine la tua lista primaria modificata.

```

lista *rmv(lista *l1, int n){

    lista *curr, *prec, *canc;
    int found = 0;

    curr = l1;
    prec = NULL;

    while(curr && !found){
        if(curr->a == n){                //Se è presente il killer
            found = 1;                  //Alza il flag
            canc = curr;                //Mettilo in canc
            curr = curr->next;           //Sostituiscilo col successivo
            if(prec!=NULL){
                prec->next = curr;       //Se ci sono ancora elementi, avanza.
            } else {
                l1 = curr;               //Hai finito.
            }
            free(canc);                 //Libera il nodo
        } else {                        //Altrimenti scorri
            prec=curr;
            curr = curr->next;
        }
    }

    return l1;
}

```

- Visualizzazione dei contenuti della lista:

```
void visualizza(lista *l1){

    while(l1 != NULL){
        printf("%d %d", l1->a, l1->b);
        l1 = l1->next;
    }
    printf("\n");
}
```

- **Distruzione della lista:**

```
lista *dtr(lista *l1){

    lista *tmp;

    while(l1 != NULL){
        tmp = l1;
        l1 = l1->next;
        free(tmp);
    }

    return NULL;
}
```

5.3 Esercizi

Ex 1: Si scriva un programma che riceve in input una sequenza di numeri dalla lunghezza indefinita che termina con il numero sentinella "-1". I numeri devono essere salvati in un array dinamico e stampati a video senza il sentinella.

Per esempio:

```
inputs: 1 5 7 9 4 -1
outputs: 1 5 7 9 4
```

Ex 2: Si scriva un programma che riceva di base un totale di 5 numeri da salvare in un array dinamico. Raggiunta la fine dello spazio, chiede all'utente se vuole aggiungere ulteriori numeri e se sì, quanti. Terminato l'inserimento bisognerà stampare a video l'array come è stato inserito e dopo con i numeri in ordine crescente.

Per esempio:

```
inputs: 1 5 7 9 4
outputs: 1 5 7 9 4 - 1 4 5 7 9
```

Ex 3: Si scriva un programma dove vengono letti da tastiera 10 interi e salvati in un array statico. Questi verranno inseriti in un array dinamico insieme alle proprie occorrenze ed infine verrà stampato quest'ultimo vettore. Usare uno struct per l'array dinamico. I numeri possono ripetersi nell'array dinamico, ma bisognerà aggiornare le loro occorrenze. Punti bonus se il programma è scomposto in funzioni.

Per esempio:

```
inputs: 1122234451
```

outputs: (1,2) (2,3) (3,1) (4,2) (5,1)(1,1)

Ex 4: Si scriva un programma capace di eseguire tutte le funzioni riguardanti le liste. La scelta della funzione da eseguire si effettua tramite un menu; da scrivere quindi: Inserimento in testa, Inserimento in coda, Ricerca di un elemento, Eliminazione di un elemento, Stampa degli elementi della lista, Distruzione della lista.

Ex 5:

Ex 6: Si scriva un programma che ricevette due sequenze di numeri di lunghezza indefinita che terminano con "-99" (Non incluso nelle sequenze), inserisce la prima nella lista L1 e la seconda in L2. Si crei una terza lista con in ordine crescente tutti i numeri di L1 ed L2, inseriti una sola volta.

Per esempio:

inputs1: 1 3 2 5 9 9 -99

inputs2: -1 4 2 7 6 -99

outputs: -1, 2, 3, 4, 5, 6, 7, 9

Ulteriori Librerie, Funzioni e QOL

6.1 Libreria ctype.h

6.2 Switch(), case, break

6.3 Argc, Argv

6.4 Divisione di un progetto su più files

I progetti che andrai a creare non saranno sempre di piccole dimensioni come in questo momento; è esattamente per questo che è necessario introdurre la strategia **Divide et Impera** al tuo modus operandi.

La prima cosa fondamentale è l'organizzazione della cartella; non puoi tenere tutto in una singola directory, nossignore. La struttura di base è sempre la seguente:

- *MainDirectory*; Cartella principale contenente l'intero programma.
 - *src*; Cartella contenente il codice sorgente.
 - *obj*; Cartella contenente il codice oggetto.
 - *bin*; Cartella contenente il codice eseguibile.
 - *Makefile*; Script che se letto dal compiler, esegue tutte le istruzioni di linking o compiling ivi scritte.
 - *README*; File testuale con spiegazioni sul codice.

Preparato l'ambiente di lavoro, è ora possibile iniziare la scrittura del codice, il quale verrà salvato nella cartella *src*. Normalmente, finita la scrittura basterebbe una semplice riga di comando a gcc per compilare il programma, ma noi dobbiamo collegare tutto; ed è qui che entra in gioco **Makefile**.

Questo file dovrà contenere tutte le istruzioni per il compilatore e viene scritto come segue:

```
all: program
program: program.o subfile.o
    gcc -o program program.o subfile.o
program.o: program.c
    gcc -c program.c
subfile.o: subfile.c
    gcc -c subfile.c
```

Risulta molto comodo perché risparmia la scrittura dei comandi da terminale, inoltre è molto semplice da modificare dovesse questa azione essere richiesta. È possibile inoltre aggiungere dei flag in cima al file per indicare i comandi da eseguire in una determinata linea; una scrittura non dissimile dall'invocazione.

Grazie ad esso è possibile direzionare i files nelle rispettive cartelle, come gli eseguibili in bin e gli oggetti in obj. Nella scrittura in C è necessaria anche una cartella *hdr*, contenente le librerie personalizzate da includere nel main. Si tratta di files dall'estensione ".h" contenenti tipi di dato e funzioni user-defined che, se opportunamente incluse, saranno utilizzabili nel main con una semplice dichiarazione o invocazione. Se questa cosa non t'attizza non so cosa possa farlo.

6.5 Debugger GDB

Soluzioni degli Esercizi

7.1 Esercizi di base

Ex 1:
Ex 2:
Ex 3:
Ex 4:
Ex 5:
Ex 6:

7.2 Condizioni e cicli

Ex 1:
Ex 2:
Ex 3:
Ex 4:
Ex 5:
Ex 6:

7.3 Vettori e Matrici

Ex 1:
Ex 2:
Ex 3:
Ex 4:
Ex 5:
Ex 6:

7.4 Funzioni e Struct

Ex 1:
Ex 2:
Ex 3:
Ex 4:
Ex 5:
Ex 6:

7.5 Stringhe e Puntatori

Ex 1:

Ex 2:

Ex 3:

Ex 4:

Ex 5:

Ex 6:

7.6 Memoria dinamica e Liste

Ex 1: Si scriva un programma che riceve in input una sequenza di numeri dalla lunghezza indefinita che termina con il numero sentinella "-1". I numeri devono essere salvati in un array dinamico e stampati a video senza il sentinella.

Per esempio:

inputs: 1 5 7 9 4 -1

outputs: 1 5 7 9 4

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

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

```
int main(){
```

```
    int *a;
```

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

```
    //Alloco lo spazio per un int.
```

```
    a = (int*) malloc(sizeof(int));
```

```
    //Failproof per sicurezza.
```

```
    if(a == NULL){
```

```
        printf("Memoria non allocata\n");
```

```
    } else {
```

```
        do {
```

```
            scanf("%d", &n);
```

```
            if(n != -1) {
```

```
                a[i] = n;
```

```
                i++;
```

```
            //Riallocazione di memoria. È posta dopo i dati già ricevuti.
```

```
            a = (int*) realloc(a, i + i * sizeof(int));
```

```
        } if(!a){
```

```
            printf("Riallocazione fallita\n");
```

```
            n = -1;
```

```
        }
```

```

        }
    } while(n != -1);

    j = i;
}

printf("Array inserito: ");
for(i = 0; i < j; i++){
    printf("%d ", a[i]);
}
printf("\n");

//Liberazione della memoria utilizzata.
free(a);
a = NULL;

return 0;
}

```

Ex 2:

Ex 3: Si scriva un programma dove vengono letti da tastiera 10 interi e salvati in un array statico. Questi verranno inseriti in un array dinamico insieme alle proprie occorrenze ed infine verrà stampato quest'ultimo vettore. Usare uno struct per l'array dinamico. I numeri possono ripetersi nell'array dinamico, ma bisognerà aggiornare le loro occorrenze. Punti bonus se il programma è scomposto in funzioni.

Per esempio:

inputs: 1122234451

outputs: (1,2) (2,3) (3,1) (4,2) (5,1) (1,1)

```

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

```

```

#define N 10

```

```

typedef struct{
int num;
int occ;
}data;

```

```

void writeArray(int *);

```

```

int main(){

    int stc[N];

    printf("Inserire 10 valori: ");
    for(int i = 0; i < N; i++){
        scanf("%d", &stc[i]);
    }
}

```

```

//Void allo scopo di mantenere il main pulito
writeArray(stc);

return 0;
}

void writeArray(int *stc){

    int i = 0;
    int j = 0;
    int count = 1;
    int tmp, size;
    data *dynamic;

    dynamic = (data*) malloc(sizeof(data));

    if(dynamic == NULL){
        printf("Errore in allocazione memoria\n");
    }

    for(i = 0; i < N; i++){
        /*tmp serve a salvare i numeri. Siccome non è inizializzato, sarà sempre
        diverso dal primo numero.*/
        if(tmp != stc[i]) {
            tmp = stc[i];
        }
        //Guarda una posizione avanti. Se uguale aumenta le occorrenze di 1.
        if(stc[i] == stc[i+1]) {
            count++;
        }
        /*Se entra in else alla prossima lettura il numero è diverso.
        Salviamo quello corrente e le occorrenze nell'array dinamico.*/
        } else {
            dynamic[j].num = tmp;
            dynamic[j].occ = count;
            j++;

            count = 1;
        }
        size = j;

        /*Riallocazione della memoria perché altrimenti esplode.
        Nel senso, non avrebbe abbastanza spazio per 10 numeri.*/
        dynamic = (data*) realloc(dynamic, size + size * sizeof(data));
    }

    printf("Array dinamico: ");
    for(int j = 0; j < size; j++){

```

```
        printf("(%d, %d) ", dynamic[j].num, dynamic[j].occ);  
    }  
    printf("\n");  
  
    free(dynamic);  
    dynamic = NULL;  
}
```

Ex 4:

Ex 5:

Ex 6: