C SHEET

compilazione

`$clang programma.c` genera file a.out
`$ .\a.out` esegue il file

altrimenti posso rinominare il file a.out con
`$ clang file.c -o nome`
`$ .\nome`

il file deve avere estensione .c

direttive pre processing

`#` : direttive del processore, a *inizio* del programma

`#define` : serve a definire una costante o una funzione
`#include` : serve a preprocessare un file con determinate funzioni

Es: `#include <stdio.h>`

- definire una costante `#define NOME valore`
- includere file `#include <nome.h>`

tipi presenti in c

char, short int, int, long int, float, double, unsigned int (anche long / short)

conversioni mediante:

- promozioni: char → short → int → long int → float → double
- cast: (tipo) variabile

particolarità:

- le stringhe sono considerate array di char
- non esiste il tipo bool, costrutti si aspettano espressioni intere (0 = falso)
- struct...

sintassi di base

- dichiarazione variabili:

- dichiarazione funzione (prototipo): `tipo_ritornato nome_funzione(lista_parametri);`
- dichiarazione di tipo:
- costrutti di controllo e cicli (if,for,while, switch)
- operatori aritmetici: +, -, *, /, %, ++, --
- operatori di confronto: < > <= >= == !=
- operatori logici: && (and), || (or), ! (not)
- costanti

costrutti
for(dichiarazione e inizio indice; condizione fine; passo): termina quando condizione fine = 0
while(condizione): se condizione uguale a 0 termina, altrimenti continua

specificatori di formato:

- `%d` : intero
- `%f` : float
- `%c` : char
- `%s` : stringa
- `%p` : puntatore
- `%x` : esadecimale
- `%2.3f` : float con almeno 2 cifre intere e 3cifre decimali (formato)

sequenze di escape:

- \n newline
- \t Tab
- \\ singola backslash '\'
- " doppi apici

struttura del programma

```
#include ...

int main(){
    blocco di codice
    return 0;
}
```

funzioni

Le funzioni vengono prima dichiarate e poi definite tramite l'implementazione

- dichiarazione: "intestazione", tipo nome(argomenti)
- implementazione: corpo

dichiarazione deve precedere la chiamata, se

variabili e passaggio per valore

le funzioni operano su **copie** dei valori degli argomenti, non sugli argomenti stessi. L'unico modo per operare sugli argomenti è passare il loro indirizzo di memoria, tramite *puntatori*.

- scope: visibilità delle variabili

array

dichiarazione `tipo nome_array[dim] = {valori}`

generalmente uso dei puntatori per accedere agli elementi dell'array o per passare l'array ad una funzione
gli array non vengono passati per valore, ma per riferimento

esempio:

```
int array[5] = {1,2,3,4,5};
int *pa = &a[0];
printf("%d", *pa); //stampa 1
printf("%d", *(pa+1)); //stampa 2
```

```
void fill(int *begin, int size, int value) {
 for(int *p=begin;p<begin+size;++p)
 *p=value;
 }
```

è possibile anche dichiarare array multidimensionali:

```
float matrix[4][3] = {{1,2,3},{4,5,6},{7,8,9},{10,11,12}};
matrix[1][1] = 2;
```

puntatori

variabile che contiene l'**indirizzo di memoria** di un'altra variabile

`tipo *nome_puntatore = valore iniziale;`

ci sono due operatori:

- adress of `&`: ottine un puntatore alla variabile (= indirizzo di memoria)
- dereferenziazione `*`: accesso alla variabile puntata dal puntatore (= valore)

tramite aritmetica dei puntatori posso accedere agli elementi di un array:
*p + i : puntatore avanza di i * sizeof(T) byte
*(p + i) : p[i]

in quanto variabili, anche i puntatori possono essere elementi di un array
`char *line[42] = {};` array di 42 puntatori a char *

NOTA:

```
int a[10][20]; //alloca spazio per 200 interi (10 x 20)
int *b[10]; //alloca spazio per 10 puntatori a intero

a[i][j] e b[i][j] denotano due int, ma b[i] può puntare a un array di lunghezza diversa
```

stringhe

le stringhe sono array di char, terminate da un carattere nullo '\0'
esempio:
`char s[] = "ciao";`
`char s[] = {'c','i','a','o','\0'};`

nota: `char *stringa = "Ciao mondo";` compila ma **NON** è corretto

librerie viste a lezione

- stdio.h: standard input/output
- string.h: funzioni per la manipolazione di stringhe

argomenti da riga di comando

un programma C può ricevere argomenti da riga di comando, tramite la funzione
`int main(int argc, char **argv)`

argc: numero di argomenti passati
argv: array di puntatori a carattere, che puntano alle stringhe degli argomenti
primo parametro: nome programma
ultimo elemento (argv[argc]) è NULL

scanf e sscanf

```
int x = 0, y = 0;
scanf("%d %d", &x, &y);
```

posso anche richiedere altri caratteri in input

```
float real = 0, float imag = 0;
scanf(" ( %f , %f )", &real, &imag);
```

leggere e convertire il valore secondo tipo specificato e ignorarlo:
`scanf("(%f %*c %f)", &real, &imag);`

allocazione dinamica

funzione malloc `void *malloc(unsigned n);` argomentto numero di byte da allocare, ritorna il puntatore all'inizio dell'area di memoria (di qualsiasi tipo)

funzione sizeof() `sizeof(tipo)` ritorna il numero di memoria dedicata per una singola istanza del tipo (es. un byte per un char e via dicendo)

la funzione free() `free(*puntatore)` serve a liberare la memoria allocata dinamicamente con malloc, in quanto la memoria allocata dinamicamente non ha uno scope preciso, rimanendo allocata

strutture

Sono un **tipo di dato** aggregato, che raggruppa variabili di tipo diverso in un'unica identità. Analoghe ai tipi base (quindi possono essere contenute in un array e possono esserci puntatori del tipo struttura)

dichiarazione di una struttura

```
struct name{
    tipo nome1;//istanza 1
    tipo nome2;//istanza 2
    ...
}

essendo come
struct name p = { componente1, componente2, ...} //varaibile di tipo struct
```

L'accesso alle componenti delle struct può avvenire in due modi:

- tramite puntatore alle componenti: `(puntatore).componente`

- stessa cosa ma tramite *operatore dedicato*: `puntatorestruttura->componente`, (s->var)

Esempio:

```
//dichiarazione di una struttura
struct point{
    float x;
    float y;
};
//dichiarazione di una variabile
struct point p = {3, 4};

//operazioni
printf("%f, %f\n", p.x, p.y);
scanf("{ %f, %f }", &p.x, &p.y);

//esempio funzione
float abs(struct point p){
    return sqrt(p.x * p.x + p.y * p.y); }

//puntatore a strutture e accesso
struct point *pp = &p;

printf("%f %f\n", pp->x, pp->y); //Equivalente alla seguente
printf("%f %f\n", (*pp).x, (*pp).y);
```

funzioni viste a lezione

getchar: `getchar()` legge carattere dallo standard input
putchar: `putchar(c)` stampa carattere nello standard output

strlen: `int strlen( *char s)` restituisce la lunghezza di una stringa
strncmp: `int strcmp( *char s1, *char s2, unsigned len)` confronto lessicografico di due stringhe (0 uguali, -1 altrimenti)
strcmp: `int strcmp( *char s1, *char s2)` versione meno sicura in quanto manca lunghezza
strncpy: `char *strncpy( *char dest, *char source, unsigned len)` copia i primi len caratteri di source in dest
strncat: `char *strncat( *char dest, *char source, unsigned len)` concatena i primi len caratteri di source a dest

printf: `printf "stringa", %1, %2, ...` stampa stringa nello standard output, sostituendo ogni % nella stringa al corrispondente argomento

scanf: `scanf("formato", &var1, &var2, ...)` legge input da standard input e lo memorizza nelle variabili passate come argomento

sscanf: `sscanf(stringa, "formato", &var1, &var2, ... )` legge da stringa fornita come parametro invece che da standard input
sprintf: `sprintf(stringa, "formato", var1, var2, ...)` stampa su una stringa invece che su standard output
snprintf: `snprintf(stringa, dimensione, "formato", var1, var2, ...)` ulteriore argomento lunghezza massima stringa (consigliata)

malloc: `void *malloc(unsigned n);` serve ad allocare n byte contigui, void * è un puntatore di qualsiasi tipo
free: `free(puntatore-malloc);` serve a liberare la memoria allocata con malloc (buona norma usarlo sempre)
realloc: `void *realloc(void *ptr, unsigned new_size);` funzione ritorna un nuovo puntatore
calloc: `void *calloc(unsigned count, unsigned size);` alloca della memoria azzerandola precedentemente