Laboratorio II

Corso A

Lezione 8

Puntatori a funzioni

Stringhe

Debugging

Puntatori a funzioni

Le funzioni in C non sono variabili

Possiamo comunque ottenere dei puntatori a funzioni definite nel programma o in librerie

I puntatori possono poi essere passati come parametro o utilizzati per invocare la funzione.

```c
#include <stdio.h>

int sum(int, int);
int mul(int,int);

int main(){

  int (*f)(int,int);

  f=sum;
  printf("3+6=%d\n", (*f)(3,6));

  f=mul;
  printf("3*6=%d\n", (*f)(3,6));
}

int sum(int x, int y){
  return x+y;
}

int mul(int x, int y){
  return x*y;
}
```

Puntatori a funzioni

Possiamo avere degli array di puntatori a funzioni.

```c
#include <stdio.h>

int sum(int, int);
int mul(int,int);

int main(){

  int (*f[2])(int,int)={sum,mul};

  printf("3+6=%d\n", (*f[0])(3,6));

  printf("3*6=%d\n", (*f[1])(3,6));
}

int sum(int x, int y){
  return x+y;
}

int mul(int x, int y){
  return x*y;
}
```

Funzione qsort

```c
int compInt(const void* a, const void * b){
  int* ac=(int*)a;
  int* bc=(int*)b;
  return *ac-*bc;

}
```

In stdlib.h

```
void qsort (void* base, size_t num, size_t size,

            int (*compar)(const void*,const void*));
```

- `base` - array
- `num` - numero di elementi
- `size` - dimensione di 1 elemento
- `compar` - puntatore a funzione per confronto

Esempio di genericità usando void*

```
> ./main
0 2 9 17 43 >
```

```c
int compInt(const void*, const void *);

int main(){

  int a[5]={17,9,2,43,0};

  qsort(a,5,sizeof(int),compInt);

  for(int i=0;i<5;i++){
    printf("%d ",a[i]);
  }
  return 0;
}
```

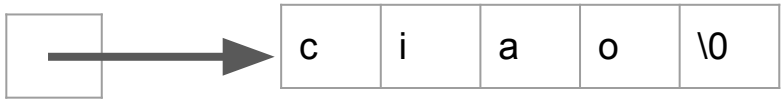
Stringhe

Array di caratteri, con ‘\0’ alla fine (carattere null, con valore 0)

char s[]=”ciao”; ->

c	i	a	o	\0

char* s=”ciao”; ->




```c
#include <stdio.h>

int main(){

  char s1[]="ciao";

  s1[2]='A';

  printf("s1=%s\n",s1);

}
```

```
> ./main
s1=ciAo
```

```c
#include <stdio.h>

int main(){

  char* s1="ciao";

  s1[2]='A';

  printf("s1=%s\n",s1);

}
```

```
> ./main
signal: segmentation fault (core dumped)
```

Stringhe

Lavorare con stringhe è analogo a lavorare con gli array.

E.g. ottenere una copia di una stringa

```c
#include <stdio.h>

int main(){

  char s1[]="ciao";
  char* s2=s1;

  s2[2]='A';

  printf("s1=%s\n",s1);
  printf("s2=%s\n",s1);

}
```

```
> ./main
s1=ciAo
s2=ciAo
```

```
> ./main
s1=ciao
s2=ciAo
```

```c
int main(){

  char s1[]="ciao";
  char s2[5];

  int i=0;
  while(s1[i]!='\0'){
    s2[i]=s1[i];
    i++;
  }
  s2[i]='\0';

  s2[2]='A';

  printf("s1=%s\n",s1);
  printf("s2=%s\n",s2);

}
```

Operazioni su stringhe

Libreria string.h include funzioni per lavorare con stringhe (qui esempi, la lista non è completa)

- `size_t strlen(const char * str)` - restituisce la lunghezza di una stringa
- `char* strcpy(char * dest, const char * src)` - copia la stringa `src` in `dest`. Restituisce `dest`.
- `int strcmp(const char * str1, const char * str2)`- confronto lessicografico, restituisce -1, 0, +1
- `char* strcat(char * dest, const char * src)` - concatena una copia di `str` alla `dest`. Restituisce `dest`.
  - Le varianti `strncpy, strncmp, strncat` - lavorano sui primi `n` caratteri

- `char* strtok(char * str , const char * delim)`

Operazioni su stringhe

`char* strtok(char * str , const char * delim)` - restituisce il pointer al prossimo token

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

int main(){

  char s[100], *tok;
  scanf("%[^\n]",s);

  tok=strtok(s,",");
  printf("%s\n",tok);

  while((tok=strtok(NULL,","))!=NULL)
    printf("%s\n",tok);

}
```

```
> ./main
12,13,14,15
12
13
14
15
```

strtok - lavora sulla stringa originale

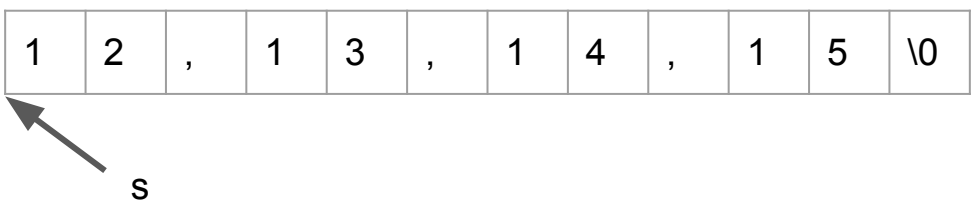
```c
int main(){

  char s[100], *tok;
  scanf("%[^\n]",s);
  printf("Stringa:%s\n",s);

  tok=strtok(s,",");
  printf("Token:%s\n",tok);

  while((tok=strtok(NULL,","))!=NULL){
    printf("Token:%s\n",tok);
  }

  printf("Stringa:%s\n",s);
}
```

```
> ./main
12,13,14,15
Stringa:12,13,14,15
Token:12
Token:13
Token:14
Token:15
Stringa:12
>
```

strtok - lavora sulla stringa originale

```c
int main(){

  char s[100], *tok;
  scanf("%[^\n]",s);
  printf("Stringa:%s\n",s);

  tok=strtok(s,",");
  printf("Token:%s\n",tok);

  while((tok=strtok(NULL,","))!=NULL){
    printf("Token:%s\n",tok);
  }

  printf("Stringa:%s\n",s);
}
```

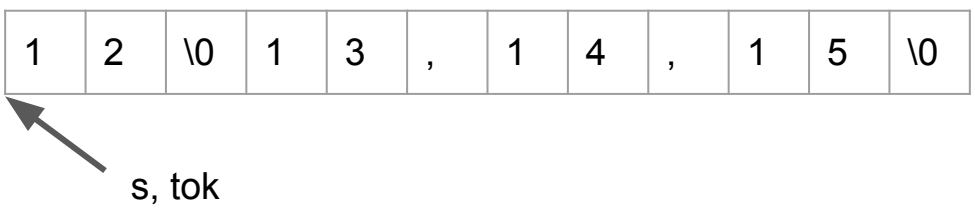
```
> ./main
12,13,14,15
Stringa:12,13,14,15
Token:12
Token:13
Token:14
Token:15
Stringa:12
>
```



strtok - lavora sulla stringa originale

```c
int main(){

  char s[100], *tok;
  scanf("%[^\n]",s);
  printf("Stringa:%s\n",s);

  tok=strtok(s,",");
  printf("Token:%s\n",tok);

  while((tok=strtok(NULL,","))!=NULL){
    printf("Token:%s\n",tok);
  }

  printf("Stringa:%s\n",s);
}
```

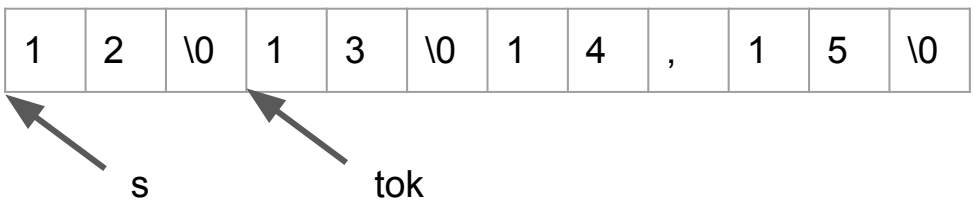
```
> ./main
12,13,14,15
Stringa:12,13,14,15
Token:12
Token:13
Token:14
Token:15
Stringa:12
>
```



strtok - lavora sulla stringa originale

```c
int main(){

  char s[100], *tok;
  scanf("%[^\n]",s);
  printf("Stringa:%s\n",s);

  tok=strtok(s,",");
  printf("Token:%s\n",tok);

  while((tok=strtok(NULL,","))!=NULL){
    printf("Token:%s\n",tok);
  }

  printf("Stringa:%s\n",s);
}
```

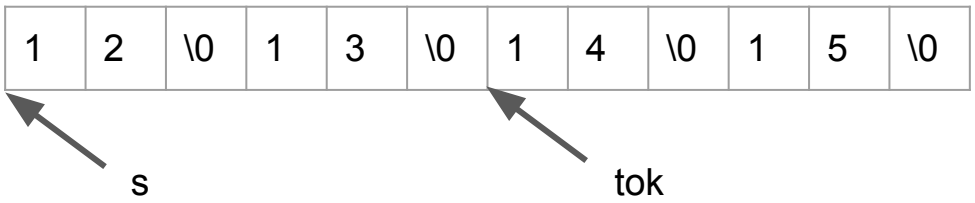
```
> ./main
12,13,14,15
Stringa:12,13,14,15
Token:12
Token:13
Token:14
Token:15
Stringa:12
>
```



strtok - lavora sulla stringa originale

```c
int main(){

  char s[100], *tok;
  scanf("%[^\n]",s);
  printf("Stringa:%s\n",s);

  tok=strtok(s,",");
  printf("Token:%s\n",tok);

  while((tok=strtok(NULL,","))!=NULL){
    printf("Token:%s\n",tok);
  }

  printf("Stringa:%s\n",s);
}
```

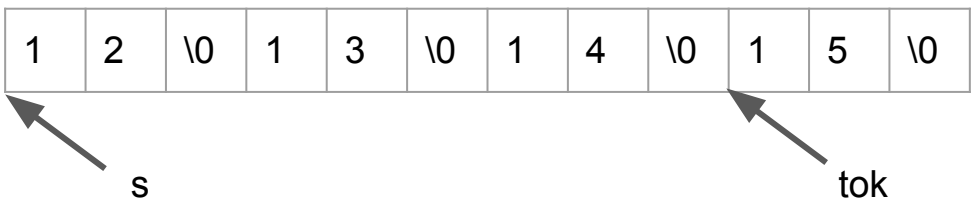
```
> ./main
12,13,14,15
Stringa:12,13,14,15
Token:12
Token:13
Token:14
Token:15
Stringa:12
>
```



strtok - lavora sulla stringa originale

```c
int main(){

  char s[100], *tok;
  scanf("%[^\n]",s);
  printf("Stringa:%s\n",s);

  tok=strtok(s,",");
  printf("Token:%s\n",tok);

  while((tok=strtok(NULL,","))!=NULL){
    printf("Token:%s\n",tok);
  }

  printf("Stringa:%s\n",s);
}
```

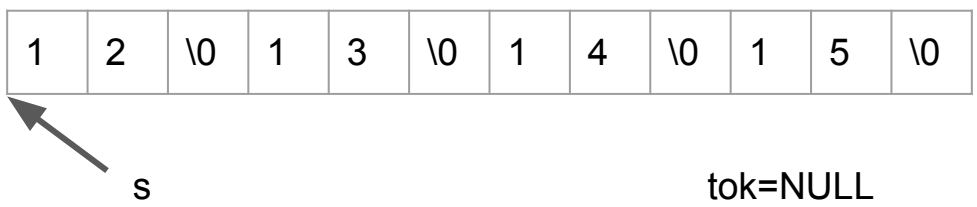
```
> ./main
12,13,14,15
Stringa:12,13,14,15
Token:12
Token:13
Token:14
Token:15
Stringa:12
>
```



strtok - lavora sulla stringa originale

```c
int main(){

  char s[100], *tok;
  scanf("%[^\n]",s);
  printf("Stringa:%s\n",s);

  tok=strtok(s,",");
  printf("Token:%s\n",tok);

  while((tok=strtok(NULL,","))!=NULL){
    printf("Token:%s\n",tok);
  }

  printf("Stringa:%s\n",s);
}
```

```
> ./main
12,13,14,15
Stringa:12,13,14,15
Token:12
Token:13
Token:14
Token:15
Stringa:12
>
```



strtok - lavora sulla stringa originale

```c
int main(){

  char *s="12,13,14,15", *tok;
  printf("Stringa:%s\n",s);

  tok=strtok(s,",");
  printf("Token:%s\n",tok);

  while((tok=strtok(NULL,","))!=NULL){
    printf("Token:%s\n",tok);
  }

  printf("Stringa:%s\n",s);
}
```

```
> ./main
Stringa:12,13,14,15
signal: segmentation fault (core dumped)
```

Operazioni su caratteri

Libreria `ctype.h` include funzioni per lavorare con caratteri (qui esempi)

`int isalpha(int)` !=0 se parametro è una lettera

`int isupper(int)` !=0 se parametro è una lettera maiuscola

`int islower(int)` !=0 se parametro è una lettera minuscola

`int isdigit(int)` !=0 se parametro è una cifra

`int isalnum(int)` !=0 se parametro è cifra o lettera

`int isspace(int)` !=0 se parametro è white space

`int toupper(int)` trasforma in lettera maiuscola se possibile, altrimenti restituisce lo stesso carattere

`int tolower(int)` trasforma in lettera minuscola se possibile, altrimenti restituisce lo stesso carattere

Leggere caratteri e stringhe

`int scanf ( const char * format, ... );` - “%c”,“%s”, “%[^\n]”, “%[^EOF]”

`int getchar ( void );` Legge un carattere da stdin

`char * fgets ( char * str, int num, FILE * stream );` Legge una riga di max num caratteri da ***stream***. Utilizzare `stdin` per leggere da standard input.

Da stringa ad altri tipi di dato

```
int sscanf ( const char * s, const char * format, ...);
```

Legge dalla stringa `s` invece che da `stdin`

```
int atoi(const char * str), long atol(const char * str),
float atof(const char * str), double atod(const char * str)
```

Trasformano una stringa in un `intero/long/float/double`

Altre funzioni per lavorare con la memoria

```
void * memset ( void * ptr, int value, size_t num );
```

Mette value in num caratteri unsigned (1 byte) a partire dalla zona di memoria puntata da ptr.

```
void * memmove ( void * destination, const void * source, size_t num );
```

Muove `num` byte dalla `source` alla `destination`.

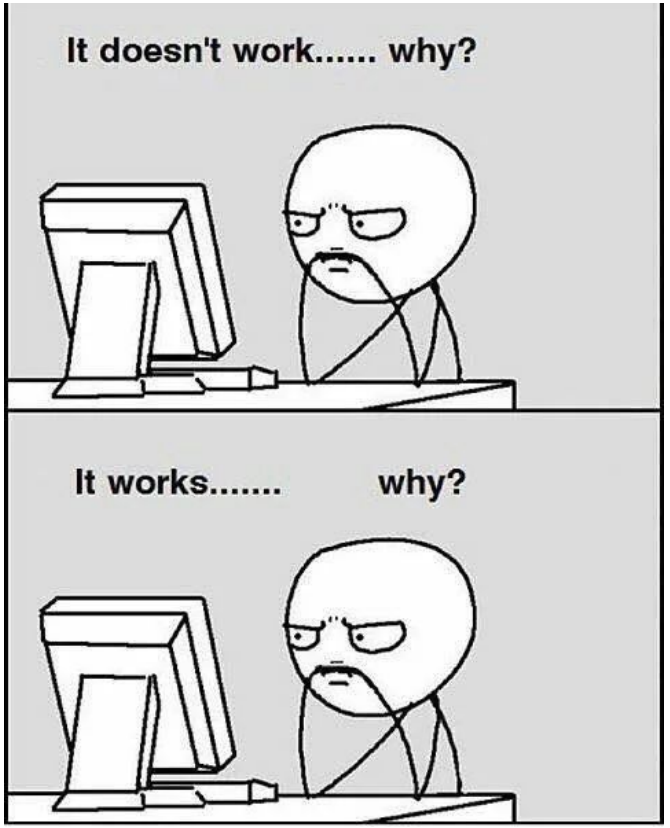
```
void * memcpy ( void * destination, const void * source, size_t num );
```

Copia `num` byte dalla `source` alla `destination`.

```
void* realloc (void* ptr, size_t size);
```

Cambia la dimensione della zona di memoria allocata in `ptr`, allocando la dimensione `size` e restituendo il puntatore (che può essere `ptr` o no).

Debugging



Debugging

- Primi tentativi tramite la stampa dei valori delle variabili
  - `printf` - stampa sulo stream `stdout`
    - buffered - la stringa non viene mostrata subito a video, ma possono essere accumulati dati nel buffer

```c
#include <stdio.h>

int main(){
  int* p=NULL;
  printf("Messaggio");
  printf("%d", *p);
}
```

```
> ./main
signal: segmentation fault (core dumped)
```

Debugging

- Soluzione: usare stream `stderr`
  - funzione `fprintf`

```c
#include <stdio.h>

int main(){
  int* p=NULL;
  fprintf(stderr,"Messaggio");
  printf("%d", *p);
}
```

```
> ./main
Messaggiosignal: segmentation fault (core dumped)
```

Assert

- In fase di sviluppo, possiamo utilizzare il macro assert per assicurarci che lo stato del programma sia corretto
- Header `assert.h`
- `void assert (int espressione)`
  - Se l'espressione è vera l'esecuzione continua
  - Se l'espressione è falsa, il programma stampa un messaggio di errore su `stderr` e finisce subito l'esecuzione.
- Per disabilitare `assert` possiamo definire il macro `NDEBUG`

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

int main(){
  int x;
  printf("inserisci un numero positivo: ");
  scanf("%d",&x);
  assert(x>0);
  printf("\nHai inserito %d\n", x);
}
```

```
inserisci un numero positivo: 7

Hai inserito 7
```

```
inserisci un numero positivo: -2
main: main.c:9: int main(): Assertion `x>0' failed.
signal: aborted (core dumped)
```

```c
#define NDEBUG
#include <stdio.h>
#include <assert.h>

int main(){
  int x;
  int a[]={1,2,3,4,5,6,7,8,9,10};
  printf("inserisci un numero positivo: ");
  scanf("%d",&x);
  assert(x>0);
  printf("\nHai inserito %d\n", x);
}
```

```
inserisci un numero positivo: -2

Hai inserito -2
```

Debugging: gdb e valgrind

- Debugging tramite la stampa dello stato delle variabili è poco efficiente e poco scalabile.
- Possiamo utilizzare tool esistenti:
  - gdb - permette di eseguire il programma passo per passo e verificare i valori delle variabili
  - valgrind - permette di analizzare la memoria allocata, identificando istruzioni illegali sulla memoria
- Non possiamo fare debug su librerie (testare prima di creare la libreria)

GDB

- Eseguo il programma in modo controllato e mi fermo per visualizzare lo stato delle variabili in un certo momento.
- Per abilitare il debugging dobbiamo compilare con opzione `-g`
  - `gcc -g -Wall -pedantic main.c -o main`
- Per avviare il debugger usare il comando
  - `gdb [opzioni] eseguibile [opzioni eseguibile]`
    - `gdb main`

GDB

- Una volta avviato, il debugger può iniziare a eseguire il programma
  - comando `run`
  - Per finire la sessione di debug usiamo comando `quit`
- Possiamo definire punti nel programma dove fermare l'esecuzione per analizzare lo stato delle variabili
  - **Breakpoint** - un punto nel codice dove fermarsi
  - **Watchpoint** - una variabile da monitorare - il programma si ferma quando la variabile cambia

GDB breakpoint

- Settare un breakpoint:
  - b (break) seguito da numero di riga o nome funzione
    - b 6 - breakpoint alla riga 6
    - b main.c:6 - breakpoint nel file main.c riga 6
    - b push - breakpoint sulla prima istruzione della funzione push
- Rimuovere un breakpoint
  - delete n - rimuove breakpoint numero n
  - enable/disable n - attiva/disattiva breakpoint n

GDB watchpoint

- Settare un watchpoint:
  - watch seguito da nome variabile
    - watch a - watchpoint su variabile chiamata a
  - Funziona solo nello scope della variabile
- Rimuovere un watchpoint
  - delete n - rimuove watchpoint numero n
  - enable/disable n - attiva/disattiva watchpoint n

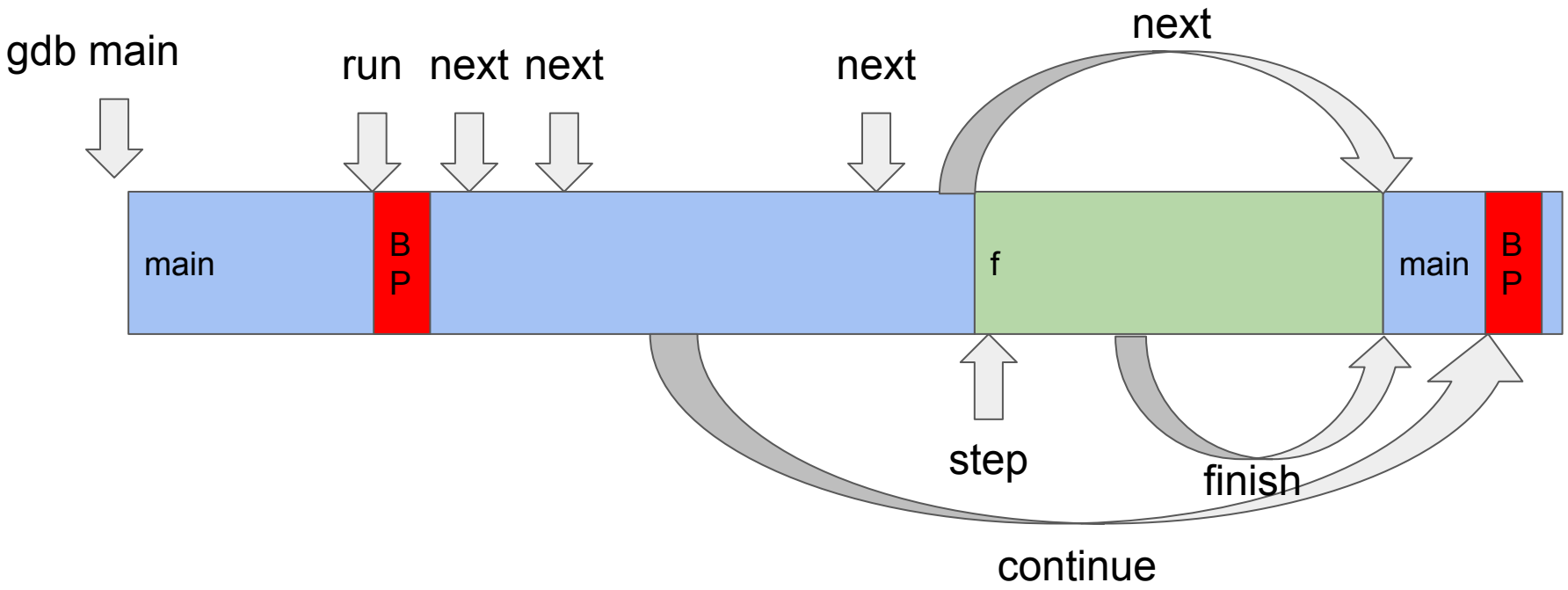
GDB - verificare lo stato dell’applicazione

- Comando `p (print)` - stampa il contenuto di una variabile
  - `p x` - stampa il valore di `x`
  - `p &x` - stampa l’indirizzo di `x`
  - `p a[3]@5` - stampa 5 valori nell’array `a` a partire dal terzo valore.
- Comando `bt (backtrace)` - stampa la pila dei record di attivazione
- Comando `x` - stampa il valore memorizzato ad un indirizzo di memoria dato

GDB - continuare l’esecuzione

- Una volta fermati su un breakpoint/watchpoint di solito si vuole continuare l’esecuzione
  - Fino al prossimo breakpoint: `continue`
  - Un’istruzione avanti:
    - senza entrare nelle funzioni : `next`
    - entrando nelle chiamate di funzioni: `step`
  - Fino all’uscita dalla chiamata di funzione corrente: `finish`

GDB - schema generale



GDB cheat sheet

GDB cheatsheet - page 1

Running

`# gdb <program> [core dump]`
Start GDB (with optional core dump).

`# gdb --args <program> <args…>`
Start GDB and pass arguments

`# gdb --pid <pid>`
Start GDB and attach to process.

`set args <args...>`
Set arguments to pass to program to be debugged.

`run`
Run the program to be debugged.

`kill`
Kill the running program.

Breakpoints

`break <where>`
Set a new breakpoint.

`delete <breakpoint#>`
Remove a breakpoint.

`clear`
Delete all breakpoints.

`enable <breakpoint#>`
Enable a disabled breakpoint.

`disable <breakpoint#>`
Disable a breakpoint.

Watchpoints

`watch <where>`
Set a new watchpoint.

`delete/enable/disable <watchpoint#>`
Like breakpoints.

<where>

`function_name`
Break/watch the named function.

`line_number`
Break/watch the line number in the current source file.

`file:line_number`
Break/watch the line number in the named source file.

Conditions

`break/watch <where> if <condition>`
Break/watch at the given location if the condition is met.
Conditions may be almost any C expression that evaluate to true or false.

`condition <breakpoint#> <condition>`
Set/change the condition of an existing break- or watchpoint.

Examining the stack

`backtrace`
`where`
Show call stack.

`backtrace full`
`where full`
Show call stack, also print the local variables in each frame.

`frame <frame#>`
Select the stack frame to operate on.

Stepping

`step`
Go to next instruction (source line), diving into function.

`next`
Go to next instruction (source line) but don't dive into functions.

`finish`
Continue until the current function returns.

`continue`
Continue normal execution.

Variables and memory

`print/format <what>`
Print content of variable/memory location/register.

`display/format <what>`
Like „print", but print the information after each stepping instruction.

`undisplay <display#>`
Remove the „display" with the given number.

`enable display <display#>`
`disable display <display#>`
En- or disable the „display" with the given number.

`x/nfu <address>`
Print memory.
n: How many units to print (default 1).
f: Format character (like „print").
u: Unit.

Unit is one of:

b: Byte,
h: Half-word (two bytes)
w: Word (four bytes)
g: Giant word (eight bytes)).

Valgrind

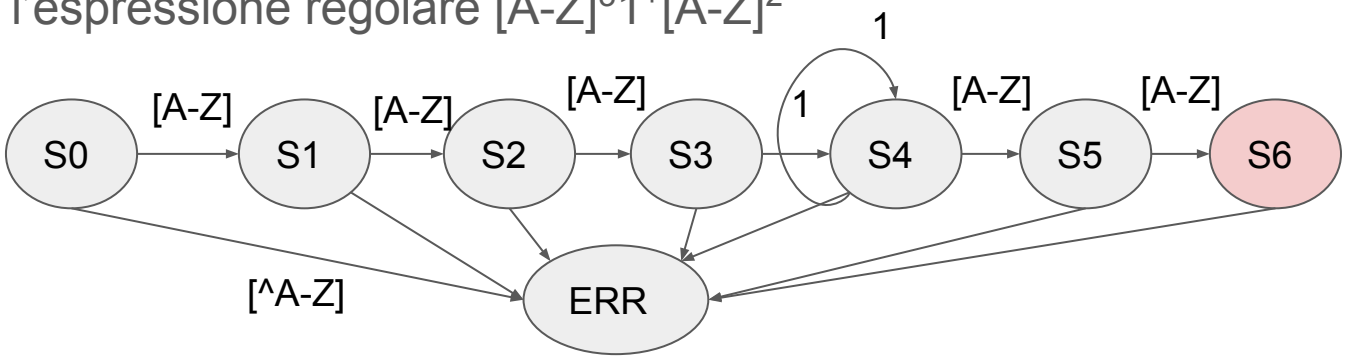
- Esegue un programma tenendo traccia di tutti gli accessi in memoria
- Stampa dei messaggi quando la memoria non è utilizzata correttamente
  - Variabili non inizializzati
  - Stack overflow
  - Memory leak

Valgrind

- Compilare il programma con opzione `-g`
- Eseguire comando
  - `valgrind ./myProg [argomenti myProg]`
- Per controllo dei memory leak usare opzione
  - `--leak-check=full`
- Per più dettagli:
  - https://valgrind.org/docs/manual/quick-start.html

Esercizi

- Il nostro strcpy.
- Sort stringhe in ordine lessicografico usando qsort.
- Il menu: word count interattivo. Il programma offre il menu: Inserimento nuova stringa, Conta parole stringa corrente, Conta caratteri stringa corrente, Conta righe stringa corrente, Esci.
- Debug - esercizi su Evo.
- Automa finito deterministico. Leggere una stringa e decidere se soddisfa l'espressione regolare $[A\text{-}Z]^3 1^+ [A\text{-}Z]^2$



Domande?