Exercise 2:
More programming Tools in Linux/UNIX

Sistemi Operativi ed in Tempo Reale

a.a. 2024/2025

Contacts

- Gabriele Penzotti
- Department of Engineering and Architecture
- Email: gabriele.penzotti@unipr.it
- Where: Pal. 1 - Ingegneria Scientifica

Outline Exercise 2

- Random number generation
- Strings in C
- Pointers
- Structure
- Socket
- Linked List

Random Number Generation

Pseudo-Random Numbers

- rand(): function for generating random number with uniform distribution on interval [0, RAND_MAX]
- srand() to set the seed of a sequence
  - sequences with the same seed are equal
  - usually initialized with time from libs time.h and stdlib.h

```
#include <stdlib.h>
#include <time.h>
int main() {
      srand(time(NULL));
      int i = rand() % 100 + 1; // random number between 1 and 100
      ...
```

Pseudo-Random Numbers

- generate a number in a specific range *[lower, upper]*

```
int i = lower + rand() % (upper - lower + 1);
```

- examples
  - random between [0, 365]

```
int i = rand() % (365 + 1);
```

  - random between [-50, +50]

```
int i = -50 + rand() % (50 - (- 50) + 1);
```

- NB: Pseudo random numbers → the sequence is deterministic based on an initial seed.

Strings in C

Strings

- **No predefined string type** in C language
- String is an **array of characters** terminated by '`\0`' (ASCII code equal to 0)
- A string of N chars is represented by

```
char str[N+1]
```

- `str` is a char pointer to the address of the first (`str[0]`) element of the string
- N is the maximum length of the string not to exceed the array limit
  - a string with L < N chars is such that: `str[L]='\0'`
  - chars after L position have undefined values

Strings

- Examples of string initialization

```
char S[6] = {'p','r','o','v','a','\0'};     // as an array of char
char S[6] = "prova";                        // as a string: implicit '\0'
```

- Example of computation of string length

```
char S[6] = "prova";
int len=0;
while(S[len] != '\0')
      len++;
printf("String %s is long %d", S, len);
```

Reading strings

- Reading string with **scanf()**
- scanf() requires pointers, but string have pointer form
  - first argument is a format string: string are identified by `"%s"`
  - second argument is a pointer

```
scanf("%s",S);
```

- Unfortunately, with scanf() strings are terminated at first occurrence of blank char " "
- Issues with strings longer than the string limit: they may not be terminated by a `'\0'`

Reading strings

- Use instead **gets()**

```
char *gets(char *S)
```

- Puts in the string pointed by S the chars read from STDIN until a char `'\n'`
- String is terminated by `'\0'`
- Returns either the pointer to the first char in the string or NULL when reading fails

```
char string [256];
printf ("Insert your full address: ");
gets (string);
printf ("Your address is: %s\n", string);
```

Strings standard functions

```
#include <string.h>
int strcmp(char *string1, char *string2) //Compare string1 e string2
char *strcpy(char *string1, char *string2) //Copy string2 in string1
int strlen(char *string) //Computes the length of a string
char *strncat(char *string1, char *string2, size_t n) //Add n chars of string2 to string1
int strncmp(char *string1, char *string2, size_t n)
                                        //Compare only the first "n“ chars of two strings
char *strncpy(char *string1, char *string2, size_t n)
                                        // Copy only the first "n“ chars of string2 in string1
```

- Example:

```
char str[MAX_SIZE];
strcpy(str,“esempio di stringa");
```

Reading strings

- C strings does not support assignment, only initializations
- **Invalid** instructions

```
char greeting[10];
greeting = "Hello";   //Cannot
```

- **Valid** instruction initialization

```
char greeting[10] = "Hello";
```

- String must be copied:

```
char greeting[10];
strcpy(greeting, "Hello");
```

NB: See example: stringhe_esempi_c.c

Pointers

Pointers and Dereferencing

- **Pointers**: variable storing the memory address of another variable
  - Pointers can access the value of the pointed variable
- Pointer **declaration**: pointer type is the type of pointed variable

```
int* pi; /* pointer to an integer variable */
char* ptr; /* pointer to a char variable */
```

- **Dereferencing**: accessing the variable value

```
*pi = 10;  /* set the value of variable pointed by pi */
```

  - pointer must be set to a correct address!

Getting the address

- Address of a variable obtained by operator ‘&’

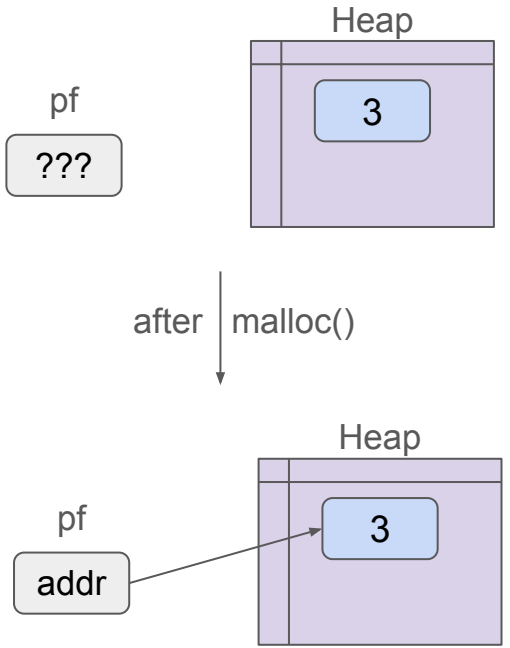
```
int i, *pi;
i = 42; /* value assignment */
pi = &i; /* pi points to address of i */
```


```
pi = 3; /* pi points to address of i */
```

Memory Allocation

- Pointers support dynamic memory allocation
- **malloc()** function for allocation
  - reserve space for allocating the desired type, e.g. float
  - returns the address of the allocated memory

```
float* pf; /* uninitialized pointer to float */
pf = (float*)malloc(sizeof(float));
```

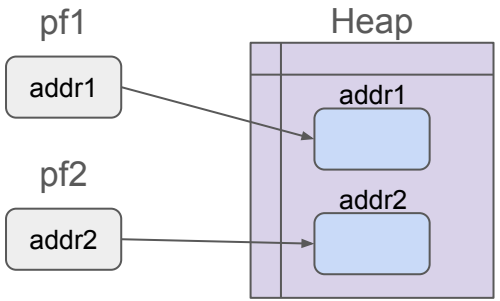


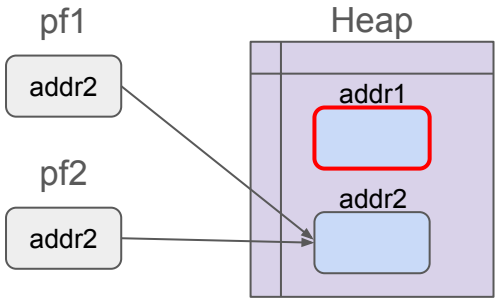
Memory Leak

- Allocated area that are not pointed and cannot be used anymore
- Example

```
pf1 = (float*)malloc(sizeof(float));
pf2 = (float*)malloc(sizeof(float));
pf1 = pf2;
```

- No reference to the area originally pointed by pf1!
  - no access to the area
- The unreferenced memory area is “garbage”

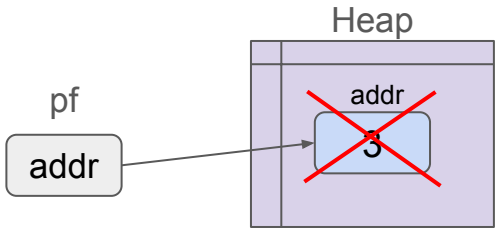




Dangling Pointer

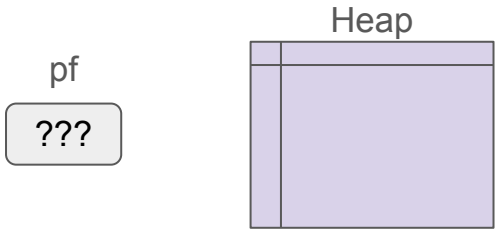
- Pointers pointing to an **invalid memory area**
  - Memory area deallocated

```
free(puntfloat);
*punfloat = 1.0f;  // error!
```



  - Memory area not yet allocated

```
float* puntfloat1; //undefined address
*punfloat = 1.0f;
```



Structure

Typedef

- Redefined **label** of an existing type
  - Hide the real type and add abstraction to the code
  - Avoid repetition of keyword struct (only for C)
- Syntax: `typedef ExistingType NewType;`

```
typedef int MyLabel;
typedef struct Node* LIST; /* pointer to first list item seen as a whole list*/
```

- Note: still used in C++, but now keyword using is recommended

```
using NewType = ExistingType;
```

Structured Types

- Composite data types consisting of heterogeneous variables
  - field: each variable of the structure
  - keyword struct
- Ex - Named struct:

```
struct Person {
      int year;
      double height;
      char name[10];
};
struct Person p1;
p1.height = 178.2;
```

- Anonymous struct + typedef:

```
typedef struct {
      int year;
      double height;
      char name[10];
} Person;
Person p1;
```

Structured Types

- Pointers to structured data and access to fields

```
typedef Person* PersonPtr;
Person alice;
PersonPtr p;
...
p = &alice;
(*p).age=5;
p->height=180.0;      // Accessing to the pointed struct
```

Socket

Socket

- **Software interface** for communicating among processed
- It associate a channel to an integer **file descriptor**
- We focus on socket type STREAM for TCP
- **Client-Server**
  - Server is active and waiting for request from clients
  - Client connects to server
  - Client and server can exchange data on established connections

- Server supports *many* simultaneous connections

Socket API

- API in C sockets (inherited from BSD UNIX sockets 1983)
  - **`socket`**: create a socket on a given domain, type and protocol
  - **`bind`**: assign a name/address to the socket
  - **`listen`**: set maximum number of accepted simultaneous connections
  - **`accept`**: server socket accept incoming connection from client (blocking API!)
  - **`connect`**: client socket request to connect to server
  - **`getsockname`**: read local address of a socket
  - **`close`**: close a file descriptor associated to a socket (used also for files, other primitives, etc.)
  - **`send`**: to send data; it is an alias of write()
  - **`recv`**: to receive data (blocking API!); it is an alias of read()

Socket: Client

- Client: setting a connection

```
#include <unistd.h>
#include <netdb.h>

struct sockaddr_in serv_addr;
struct hostent* server;
char* host_name = "127.0.0.1";            /* address of the server as string*/
int port = 8000;
if ((server = gethostbyname(host_name)) == 0 ) {           /* address */
      perror("Error resolving local host\n");
      exit(1);
}
bzero(&serv_addr, sizeof(serv_addr));
serv_addr.sin_family = AF_INET;
serv_addr.sin_addr.s_addr = ((struct in_addr *)(server->h_addr))->s_addr;
serv_addr.sin_port = htons(port);   /* htons() handle little and big endians*/
```

info about server

Clear serv_addr

Specify ipv4 family

Extracts and assigns the server's IP address.

Socket: Client

Protocol Family internet (ipv4)
Socket type Stream
Default Protocol: TCP

- Client: setting a connection

```
int sockfd = socket(PF_INET, SOCK_STREAM, 0); /* file descriptor sockfd */
if ( sockfd == -1 ) {
      … /* error */
}
if (connect(sockfd, (void*)&serv_addr, sizeof(serv_addr) ) == -1) {
      … /* error */
}
```

- Hence after connect is established: communication with `send()` and `recv()` until the **sockfd** is closed

Socket: Server

- Server: opening and waiting for client connections (this part is similar for server)

```
#include <unistd.h>
#include <netdb.h>

struct sockaddr_in serv_addr;
struct sockaddr_in cli_addr;
int sockfd = socket(PF_INET, SOCK_STREAM, 0);
if ( sockfd == -1 ) { ... }
bzero(&serv_addr, sizeof(serv_addr));
serv_addr.sin_family = AF_INET;
serv_addr.sin_addr.s_addr = INADDR_ANY;          /* the address is the local one */
serv_addr.sin_port = htons(port);                /* htons() little and big endians*/

/* bind() associates the address to the socket */
if (bind(sockfd, (struct sockaddr *)&serv_addr, sizeof(serv_addr)) == -1) { ... }
```

Accept connections on any available network interface

Bind socket to address and port

Socket: Server

```
/* maximum number of connection kept in the socket queue*/
if (listen( sockfd, 20 ) == -1) { … }
socklen_t address_size = sizeof( cli_addr );
```

- Main server **loop** (infinite in this case)

```
while(1) {
      /* new connection acceptance */
      int newsockfd = accept( sockfd, (struct sockaddr *)&cli_addr, &address_size);

      if (newsockfd == -1) { ... }
            // send/recv until close; newsockfd must be stored to communicate on connection
      }
```

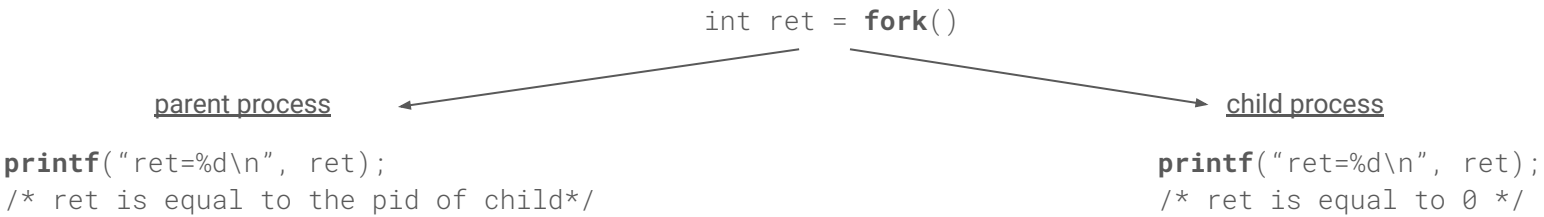
Accepts a new client connection and creates a new socket (*newsockfd*) for it

Fork

- UNIX/Linux API for starting a new process

```
# include <unistd.h>
int fork(void);
```

- An identical (child) process is created: same code and memory content of parent process
- Only difference: the **return value** of fork() is **0** for child **PID** value of the child for parent



Fork

- The return value can be used to **differentiate** the code executed by father and child processes

```
int ret = fork();
if (ret == 0) {
      /* code executed by child */
}
else {
      /* code executed by father */
}
```

- After the fork() the two processes are independent, i.e. do not share memory

Concurrent Server

- Creates a **child** process to every **request**: server not blocked by serving a client

```
int main(int argc, char *argv[]) {
       ...    /* Initialize socket */

       while (1) {
              ...
              if ( (conn_fd = accept(list_fd, (struct sockaddr *)&client, &len)){
                     perror("accept error");    exit(-1);
              }
              /* Fork to handle connection */
              if ( (pid = fork()) < 0 ){
                     perror("fork error");      exit(-1);
              }
              if (pid == 0) { /* child servicing the request from conn_fd */ }
              else { /* father code */ }
       }
```

Linked List

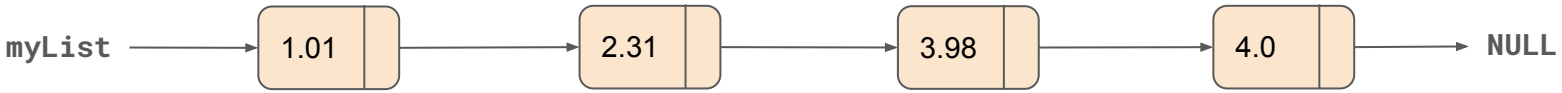
Linked List

- SORT exercises require a “container” storing items
  - arbitrary and dynamic size of container
- Linked list: simplest dynamic data structure in C language
- Each Node is divided in two parts
  - data storage
  - reference to next item

```
/* Data contained in item ***/
typedef struct {
	double value;	/*** esempio ***/
} itemType;

/* Node of the list */
struct LINKED_LIST_NODE {
	itemType item;
	struct LINKED_LIST_NODE *next;
};

/* Alias for the node */
typedef struct LINKED_LIST_NODE NODE;

/* Pointer to first item of the list represents
the whole list! */
typedef struct NODE * LIST;
```

Linked List

- Example



- List is represented by a pointer to the first node (e.g., myList)
- Field **next** of last node point to NULL

Linked List Code

```
/* Constructor */
LIST NewList(); /* Initialization */

/* Selector */
itemType getHead(LIST l);  /* Returns first item l*/
itemType getTail(LIST l);  /* Returns last item */

/* Predicates */
#define BOOL int
BOOL isEmpty( LIST l );    /* TRUE if list is empty */
int getLength( LIST l );   /* computes the length of list */
itemType * Find( LIST l, itemType item ); /* Finds an element if it exists; otherwise returns NULL */
```

Linked List Code

```
/* Insertion and removal of items */
LIST EnqueueFirst(LIST l, itemType item );     /* Inserts item in first position */
LIST EnqueueLast(LIST l, itemType item );      /* Inserts item in last position */
LIST EnqueueOrdered(LIST l, itemType item );   /* inserts element in order */
LIST DequeueLast( LIST l );       /* Removes the last item */
LIST DequeueFirst( LIST l );      /* Removes the first item */
LIST Dequeue(LIST l, itemType item );   /* Removes a given item if it is in the list */

/* Destructors */
LIST DeleteList( LIST l );

/* Other */
void PrintList(LIST l);    /* Prints list */
void PrintItem(itemType item);
```

Linked List Code

```
/* Costructor */

LIST NewList(){
       return NULL;
}


NODE * createNode( itemType item )
{
       NODE * p = (NODE*)malloc(sizeof(NODE));
       assert( p != NULL );
       p->item = item;
       p->next = NULL;
       return p;
}
```

```
/* alternative implementation */
NODE *createElement(itemType item) {
       NODE *p;
       p = (NODE *)malloc(sizeof(NODE));
       if (isEmpty(p)) {
              printf("create element failed.\n");
              exit(0);
       }
       p->item = item;
       p->next = NULL;
       return p;
}
```

Assert

- Macro assert() evaluates an expression and terminates if not true
- Used to check invariants, i.e. properties that must hold during execution
- if an expression it fails unexpectedly there is a potential bug in the code or a function is wrongly used

```
#include <assert.h>
...
assert(x > 0);
```

- Possible implementation

```
#define MY_ASSERT(X)  if (!(X)) { printf("failed %s\n",#X); exit(-1); }
```

Linked List Code

```
/* Predicates */

BOOL isEmpty( LIST l ){          /* check if list is empty */
      return (l == NULL);
}

int getLength(LIST l){           /* computes the length of list */
      int size = 0;
      LIST tmp = l;
      while ( !isEmpty(tmp) ){
            ++size;
            tmp = tmp -> next;
      }
      return size;
}
```

Linked List Code

```
/* Selectors */

itemType getHead(LIST l){        /* Returns item to first item in list l */
       assert( !isEmpty(l) );
       return l->item;
}


itemType getTail(LIST l){        /* Returns item to last item in list l */
       NODE * tmp = l;
       assert( !isEmpty(l) );

       while( !isEmpty(tmp->next) )
              tmp = tmp->next;

       return tmp->item;
}
```

Linked List Code

```
/* Insertion */

LIST EnqueueLast (LIST l, itemType item){            /* Inserts item to last position in the list */
      NODE * new_node = createNode(item);
      if ( isEmpty( l ) ) {                  /* empty list: item in head position */
            l = new_node;
      }
      else{
      LIST tmp = l;
      while ( !isEmpty( tmp -> next ) )
            tmp = tmp -> next;
            tmp -> next = new_node;}

      return l;
}
```

Linked List Code

```
/* Removal */
LIST Dequeue( LIST l, itemType item ) {        /* Removes the given item from list if exist in list */
      if (!isEmpty( l )) {
            if ( itemCompare( l -> item, item ) == 0 ) {   /* remove item from head */
                  NODE * todel = l;
                  l = l -> next;
                  deleteNode( todel );
            } else {
            LIST tmp = l;
            while ( !isEmpty(tmp ->next) && itemCompare(tmp->next->item, item ) != 0 )
                  tmp = tmp -> next;
            if ( !isEmpty( tmp -> next ) ){         /* if item is found, then it is removed */
                  NODE * todel = tmp -> next;
                  tmp -> next = tmp -> next -> next;
                  deleteNode( todel );
            }
      }
}
return l;
}
```

Loops until:
- The next node is NULL (end of the list)
- The **next** node contains the matching item

Linked List Code

- `itemCompare` - General comparison function inspired by `strcmp()`:
  - `return value >0 if item1 > item2;`
  - `return value <0 if item1 < item2;`
  - `return value == 0 if item1 == item2.`
- This function is used to sort, search or manage order → It must be adapted to each exercise

```
int itemCompare( itemType item1, itemType item2 ){
      if ( item1.value > item2.value )        /* example with float field */
            return 1;
      else if ( item1.value < item2.value )
            return -1;
      else
            return 0;
}
```

Linked List Code

```
/* Destructor */

void deleteNode( NODE * p ){      /* frees node p */
      free(p);
}

LIST DeleteList( LIST l )  {      /* deallocate all the nodes */
      LIST tmp = l;
      while ( !isEmpty(tmp) )
      {
            NODE * todel = tmp;
            tmp = tmp -> next;
            deleteNode( todel );
      }                           /* all node visited and freed */
      return NewList();
}
```

Linked List Code

```
/* Print list */

void PrintList( LIST l ){
      LIST tmp = l;
      while (!isEmpty(tmp)) {
            PrintItem( tmp->item );
            tmp = tmp->next;
            if ( ! isEmpty(tmp) )
                  printf("\n");
      }
}
```

- Custom `PrintItem()`: it depends on the item used in your problem

Exercises

Exercise: socket

- Try to execute a server process on one terminal, and executes different clients in different terminals
  - If you want, modify the server or client behavior

- **Exercise**: Implement a Client that perform multiple concurrently requests
  - Each Client must create multiple concurrent instances.
  - Each instance generates a unique number and sends it to the server, receives a response and prints the sent and the received number.
  - Server simply take the number and add 1.
  - TIPS:
    - unique seed: `srand(time(NULL) + ??? )`
    - use `sleep()` to see client child processes prints (or loop `wait()`)

Exercise: List

- Implement the following functions

```
LIST EnqueueFirst(LIST l, itemType item );     /* Inserts item in first position of list */

LIST DequeueLast( LIST l );      /* Removes the last item from list, if list is not empty */

LIST EnqueueOrdered(LIST l, itemType item );   /* Inserts the item in the list according to an order */

itemType * Find( LIST l, itemType item );
      /* Finds given item in the list and returns a pointer to the item
            (note: pointer to the item, not to the node type!!!) */
```

Problem Solving

- Decompose your exercise into functions
- Possible criteria:
  - Decompose the problem into smaller functions
  - A function must solve a single (sub)problem independent form the other parts
  - It must be clear what a function does
  - Keep the function as short as possible

Problem Solving

- Minimum numbers of function parameters required to solve the problem
- No constraints on parameters
- If number of parameters is high, possible warning
  - use a structure to store parameters?
- If you are doing cut & paste of parts of your code, then you may need a function doing it instead
- Avoid non-local access to variables:
  - access to data external to function only through parameters (with few exceptions)
  - violation of black box principle (only in very specific cases)