Preparazione 2

Esercizio 1

```
// Funzione C

#include "e1.h"

// v è un array, n è la lunghezza dell'array
void clear(short* v, unsigned n) {
    unsigned i=0;
    while (i<n) v[i++]=0;
}


// C Equivalente

void clear(short* v, unsigned n){
    short* eax = v;
    unsigned ecx = 0;
L:  if (ecx >= n) goto E;
    eax[ecx] = 0;
    ecx++;
    goto L;

E:  return;
}

// ASM

.global clear

clear:

    movl 4(%esp), %eax;
    xorl %ecx, %ecx

L:  cmpl 8(%esp), %ecx
    jae E
    movw $0, (%eax, %ecx, 2)
    incl %ecx
    jmp L

E:  ret
```

Esercizio 2

```
// Funzione C

#include "e2.h"

// v è un array, n è la lunghezza dell'array
void clear2(short* v, unsigned n) {
    short* p=v+n;
    while (v<p) *v++=0;
}


// C Equivalente

#include "e2.h"

void clear2(short* v, unsigned n){
    short* eax = v;
    short* ecx = v;
    eax = eax + n;
L:  if (ecx >= eax) goto E;
    *ecx = 0;
    ecx++;
    goto L;

E:  return;
    }

// ASM

.global clear2

clear2:

    movl 4(%esp), %eax
    movl 4(%esp), %ecx
    addl 8(%esp), %eax

L:  cmpl %eax, %ecx
    jge E
    movl $0, (%ecx)
    incl %ecx
    jmp L

E:  ret
```

Esercitazione 1

Esercizio 4

Scrivere una versione personale della funzione della libreria standard libc `strcat` che concatena alla stringa `dest` la stringa `src`.

```c
char *my_strcat(char *dest, const char *src) {
    char* concat = dest;
    while (*dest){
        dest++;
    }

    while (*src){
        *dest = *src;
        src++;
        dest++;
    }
    return concat;
}
```

Esercitazione 2

Esercizio 3

```c
// Funzione C

char my_strcmp(const char* s1, const char* s2) {
    while (*s1 && *s1 == *s2) {
        s1++;
        s2++;
    }
    return *s1 - *s2;
}
```

```c
// C Equivalente

char my_strcmp(const char* s1, const char* s2){
    const char* ecx = s1;
    const char* edx = s2;

L:  if (*ecx == 0) goto E;
    const char* eax = *ecx;
    if (eax != *edx) goto E;
    ecx++;
    edx++;
    goto L;

E:;
    eax = *ecx;
    eax = eax - *edx;
    return eax;
    }
```

```asm
// ASM

.global my_strcmp

my_strcmp:

    movl 4(%esp), %ecx
    movl 8(%esp), %edx

L:  cmpb $0, (%ecx)
    je E
    movb (%ecx), %al
    cmpb (%edx), %al
    jne E
    incl %ecx
    incl %edx
    jmp L

E:  movb (%ecx), %al
    subb (%edx), %al
    ret
```

Esercizio 4

Scrivere una funzione che, data una stringa `s` e una stringa `sub`, calcola il numero di posizioni distinte in `s` in cui `sub` appare come sottostringa.

```c
int aux(const char* str1, const char* str2){
    while(*str1++ == *str2++)
    if (!(*str2))
        return 1;
    return 0;
    }


int count_substrings(const char* s, const char* sub){
    int conta = 0;

    if (*s == '\0' && *sub == '\0')
        return 1;

    while(*s){
        if (*s == *sub)
            conta += aux(s, sub);
        s++;
        }
    return conta;
    }
```

Preparazione 3

Header

```c
#ifndef __NODE_T__
#define __NODE_T__

typedef struct node_t {    // base
    short elem;            // offset: 0 |xx..|   (base)
    struct node_t *next;   // offset: 4 |xxxx|  4(base)
} node_t;                  // sizeof: 8

void list_new(node_t **l);
int  list_new_from_buffer(node_t **l, short* buf, int n);
void list_delete(node_t **l);
int  list_add_first(node_t **l, short elem);
int  list_sum(const node_t *l);

#endif
```

Esercizio 1

```c
// Funzione C

#include <stdlib.h>
#include "e1.h"

void list_new(node_t **l) {
    *l = NULL;
}
```

```c
// C Equivalente

#include <stdlib.h>
#include "e1.h"

void list_new(node_t** l){
    node_t** eax = l;
    *eax = NULL;
    }
```

```asm
// ASM

.global list_new

list_new:

    movl 4(%esp), %eax
    movl $0, (%eax)
    ret
```

Esercizio 2

```c
// Funzione C

#include <stdlib.h>
#include "e2.h"

int list_sum(const node_t *p) {
    int s = 0;
    for (; p!=NULL; p=p->next)
        s += p->elem;
    return s;
}
```

```c
// C Equivalente

#include <stdlib.h>
#include "e2.h"

int list_sum(const node_t* p){
    int eax = 0;
    const node_t* ecx = p;
L:  if (ecx == 0) goto E;
    eax = eax + (*ecx).elem;
    ecx = (*ecx).next;
    goto L;
E:  return eax;
    }
```

```asm
// ASM

.global list_sum

list_sum:

    xorl %eax, %eax
    movl 4(%esp), %ecx

L:  cmpl $0, %ecx
    je E
    addl (%ecx), %eax
    movl 4(%ecx), %ecx
    jmp L

E:  ret
```

Esercizio 3

```c
// Funzione C

#include "e3.h"

void merge(const int* a, const int* b, int* c, int na, int nb) {
    int i=0, j=0, k=0;
    while(i<na && j<nb)
        if (a[i]<=b[j]) c[k++] = a[i++];
        else            c[k++] = b[j++];
    while(i<na) c[k++] = a[i++];
    while(j<nb) c[k++] = b[j++];
}
```

```c
// C Equivalente

#include "e3.h"

void merge(const int* a, const int* b, int* c, int na, int nb){
    int eax = 0;    // i
    int ecx = 0;    // j
    int edx = 0;    // k
    const int* ebx = a;
    const int* esi = b;
    int* edi = c;

L1: if (eax >= na) goto L2;
    if (ecx >= nb) goto L2;
    if (ebx[eax] <= esi[ecx]) goto IF;
    goto EL;

IF: edi[edx] = ebx[eax];
    edx++;
    eax++;
    goto L1;

EL: edi[edx] = esi[ecx];
    edx++;
    ecx++;
    goto L1;

L2: if (eax >= na) goto L3;
    edi[edx] = ebx[eax];
    edx++;
    eax++;
    goto L2;

L3: if (ecx >= nb) goto E;
    edi[edx] = esi[ecx];
    edx++;
    ecx++;
    goto L3;

E:  return;
    }
```

```
// ASM

.global merge

merge:

    pushl %ebx
    pushl %edi
    pushl %esi

    xorl %eax, %eax                 # eax <-> i
    xorl %ecx, %ecx                 # ecx <-> j
    xorl %edx, %edx                 # edx <-> k
    movl 24(%esp), %edi             # edi <-> b

L1: cmpl 28(%esp), %eax             # eax >= na
    jge L2
    cmpl 32(%esp), %ecx             # ecx >= nb
    jge L2
    movl 20(%esp), %ebx             # ebx <-> b
    movl (%ebx, %ecx, 4), %esi      # esi = ebx[ecx]
    movl 16(%esp), %ebx             # ebx <-> a
    cmpl %esi, (%ebx, %eax, 4)      # ebx[eax] <= esi
    jle IF
    jmp EL

IF: movl (%ebx, %eax, 4), %esi      # esi = ebx[eax]
    movl %esi, (%edi, %edx, 4)      # edi[edx] = esi
    incl %edx
    incl %eax
    jmp L1

EL: movl 20(%esp), %ebx             # ebx <-> b
    movl (%ebx, %ecx, 4), %esi      # esi = ebx[ecx]
    movl %esi, (%edi, %edx, 4)      # edi[edx] = esi
    incl %edx
    incl %ecx
    jmp L1

L2: cmpl 28(%esp), %eax             # eax >= na
    jge L3
    movl 16(%esp), %ebx             # ebx <-> a
    movl (%ebx, %eax, 4), %esi      # esi = ebx[eca]
    movl %esi, (%edi, %edx, 4)      # edi[edx] = esi
    incl %edx
    incl %eax
    jmp L2

L3: cmpl 32(%esp), %ecx             # ecx >= nb
    jge E
    movl 20(%esp), %ebx             # ebx <-> b
    movl (%ebx, %ecx, 4), %esi      # esi = ebx[ecx]
    movl %esi, (%edi, %edx, 4)       # edi[edx] = esi
    incl %edx
    incl %ecx
    jmp L3

E:
    popl %esi
    popl %edi
    popl %ebx
    ret
```

Esercitazione 3

Esercizio 1

```c
// Funzione C

int fib(int n) {
    if (n<2) return 1;
    return fib(n-1)+fib(n-2);
}
```

```c
// C Equivalente

int fib(int n){
    int ecx = n;
    int eax = 1;
    if (ecx < 2) goto E;
    ecx = ecx-1;
    eax = fib(ecx);
    int edx = eax;
    ecx = ecx -1;
    eax = fib(ecx);
    eax = eax + edx;

E:  return eax;
    }
```

```asm
// ASM

.global fib

fib:

    pushl %edi
    pushl %esi
    subl $4, %esp

    movl 16(%esp), %edi
    movl $1, %eax
    cmpl $2, %edi
    jl E
    decl %edi
    movl %edi, (%esp)
    call fib
    movl %eax, %esi
    decl %edi
    movl %edi, (%esp)
    call fib
    addl %esi, %eax

E:  addl $4, %esp
    popl %esi
    popl %edi
    ret
```

Esercizio 3

```c
// Funzione C

#include <stdlib.h>
#include <string.h>

void* clone(const void* src, int n) {
    void* des = malloc(n);
    if (des==0) return 0;
    memcpy(des, src, n);
    return des;
}
```

```c
// C Equivalente

#include <stdlib.h>
#include <string.h>

void* clone(const void* src, int n){
    int ecx = n;
    void* eax = malloc(ecx);
    if (eax == 0) return 0;
    eax = memcpy(eax, src, ecx);
    return eax;
}
```

```asm
// ASM

.global clone

clone:

    pushl %ebx
    subl $12, %esp

    movl 24(%esp), %ebx
    movl %ebx, (%esp)
    call malloc
    cmpl $0, %eax
    je E

    movl %eax, %ebx
    movl %ebx, (%esp)
    movl 20(%esp), %ebx
    movl %ebx, 4(%esp)
    movl 24(%esp), %ebx
    movl %ebx, 8(%esp)
    call memcpy

E:
    addl $12, %esp
    popl %ebx
    ret
```

Preparazione 4

Header

```c
#ifndef __SWAP_STRUCT__
#define __SWAP_STRUCT__

typedef struct {
    char buf[5];    // [xxxx] + [x...]
    int n;          // [xxxx]
    char enabled;   // [x...]
} buf_t;            // sizeof(struct) = 16

void swap(buf_t *b1, buf_t *b2);

#endif
```

Esercizio 1

```c
// Funzione C

#include "e1.h"

void swap(buf_t *b1, buf_t *b2) {
    buf_t tmp = *b1;
    *b1 = *b2;
    *b2 = tmp;
}
```

```c
// C Equivalente

#include <string.h>
#include "/e1.h"

void swap(buf_t *b1, buf_t *b2) {
        buf_t *ebx = b1;
        buf_t *edi = b2;
        buf_t tmp;
        memcpy(&tmp, b1, sizeof(buf_t));
        memcpy(b1, b2, sizeof(buf_t));
        memcpy(b2, &tmp, sizeof(buf_t));
}
```

```
// ASM

.global swap

swap:
        pushl %ebx
        pushl %edi
        pushl %esi
        subl $28, %esp          # 16 per la variabile tmp + 12 per i parametri di memcpy

        movl 44(%esp), %ebx     # buf_t *ebx = b1;
        movl 48(%esp), %edi     # buf_t *edi = b2;

        leal 12(%esp), %esi     # 12(%esp) e' &tmp (%esi = &tmp)
        movl %esi, (%esp)       # &tmp e' il primo parametro di memcpy
        movl %ebx, 4(%esp)      # b1 e' il secondo parametro di memcpy
        movl $16, 8(%esp)       # sizeof(buf_t) = 16 e' il terzo parametro di memcpy
        call memcpy             # memcpy(&tmp, b1, sizeof(buf_t));

        movl %ebx, (%esp)       # b1 e' il primo parametro di memcpy
        movl %edi, 4(%esp)      # b2 e' il secondo parametro di memcpy
        call memcpy                 # memcpy(b1, b2, sizeof(buf_t));

        movl %edi, (%esp)       # b2 e' il primo parametro di memcpy
        movl %esi, 4(%esp)      # &tmp e' il secondo parametro di memcpy
        call memcpy             # memcpy(b2, &tmp, sizeof(buf_t));

        addl $28, %esp
        popl %esi
        popl %edi
        popl %ebx
        ret
```

Esercitazione 4

Header -> Preparazione 3

Esercizio 1

```c
// Funzione C

#include <stdlib.h>
#include "e1.h"

int list_add_first(node_t **l, short elem) {
    node_t *p = *l;
    node_t *n = malloc(sizeof(node_t));
    if (n == NULL) return -1;           // allocation error
    n->elem = elem;
    n->next = p;
    *l = n;
    return 0;
}
```

```c
// C Equivalente

#include <stdlib.h>
#include "e1.h"

int list_add_first(node_t** l, short elem){
    node_t** esi = l;
    node_t* ebx = *esi;
    node_t* eax = malloc(8);
    if (eax == NULL) return -1;
    node_t* ecx = eax;
    (*ecx).elem = elem;
    (*ecx).next = ebx;
    *esi = ecx;
    return 0;
}
```

```
// ASM

.global list_add_first

list_add_first:

    pushl %ebx
    pushl %esi
    subl $4, %esp

    movl 16(%esp), %esi
    movl (%esi), %ebx
    movl $8, (%esp)
    call malloc
    cmpl $0, %eax
    je E
    movw 20(%esp), %dx
    movw %dx, (%eax)
    movl %ebx, 4(%eax)
    movl %eax, (%esi)
    xorl %eax, %eax
    addl $4, %esp
    popl %esi
    popl %ebx
    ret

E:  movl $-1, %eax
    addl $4, %esp
    popl %esi
    popl %ebx
    ret
```

Esercizio 2

```c
// Funzione C

#include <stdlib.h>
#include "e2.h"

int list_equal(const node_t *l1, const node_t *l2) {
    while (l1!=NULL && l2!=NULL) {
        if (l1->elem != l2->elem) return 0;
        l1 = l1->next;
        l2 = l2->next;
    }
    return l1==NULL && l2==NULL;
}
```

```c
// C Equivalente

#include <stdlib.h>
#include "e2.h"

int list_equal(const node_t* l1, const node_t* l2){
    const node_t* ecx = l1;
    const node_t* edx = l2;
L:  if (ecx == NULL) goto E;
    if (edx == NULL) goto E;

    if ((*ecx).elem != (*edx).elem) return 0;
    ecx = (*ecx).next;
    edx = (*edx).next;
    goto L;

E:  if (ecx != NULL) goto F;
    if (ecx != NULL) goto F;
    return 1;

F:  return 0;
}
```

```
// ASM

.global list_equal

list_equal:

    pushl %ebx

    movl 8(%esp), %ecx
    movl 12(%esp), %edx

L:  cmpl $0, %ecx
    je E
    cmpl $0, %edx
    je E
    movw (%edx), %bx
    cmpw %bx, (%ecx)
    jne F
    movl 4(%ecx), %ecx
    movl 4(%edx), %edx
    jmp L

E:  cmpl $0, %ecx
    jne F
    cmpl $0, %edx
    jne F
    movl $1, %eax

    popl %ebx
    ret


F:  xorl %eax, %eax
    popl %ebx
    ret
```

Esercizio 3

Scrivere un'implementazione della funzione standard C `strpbrk`: la funzione deve restituire il puntatore alla prima occorrenza in `s1` di un qualsiasi carattere presente nella stringa `s2` oppure `NULL` se alcun carattere di `s2` appare in `s1` prima che questa stessa termini.

Esempio: Se `s1 = "Once again, this is a test"` e `s2 = "ftir"`, la funzione deve restituire il puntatore alla prima occorrenza in `s1` di un qualsiasi carattere nella stringa `s2`, cioè il puntatore alla `i` di `"again"`.
Se `s2 = ":#F"`, allora la funzione deve restituire `NULL`.

```c
#include <stdlib.h>
#include <stdio.h>
#include "e3.h"

char* my_strpbrk(const char* s1, const char* s2){
    const char* temp;
    while (*s1){
        temp = s2;
        while(*temp){
            if (*s1 == *temp)
                return (char*) s1;
            temp++;
            }
        s1++;
        }
    return NULL;
}
```

Preparazione 5

Header

```c
typedef struct {
    int quot;   // [xxxx] 4
    int rem;    // [xxxx] 4
} div_rem_t;    // sizeof(struct) = 8

void array_div(const int *a, const int *b, div_rem_t *res, int n);
```

Esercizio 1

```c
// Funzione C

#include "e1.h"

void array_div(const int *a, const int *b, div_rem_t *res, int n) {
    int i;
    for (i=0; i<n; ++i) {
        int min = a[i] <  b[i] ? a[i] : b[i];
        int max = a[i] >= b[i] ? a[i] : b[i];
        res[i].quot = max / min;
        res[i].rem  = max % min;
    }
}
```

```c
// C Equivalente

#include "e1.h"

void array_div(const int* a, const int* b, div_rem_t* res, int n){
        int ebx = 0;
        const int* ecx;
        const int* esi;
        int eax;
        int edi;
        int edx;
L:      if (ebx >= n) goto E;
        ecx = a;
        esi = b;

        if (ecx[ebx] >= esi[ebx]) goto F;
        eax = esi[ebx];
G:
        if (ecx[ebx] < esi[ebx]) goto H;
        edi = esi[ebx];
I:;
        int temp = eax;
        eax = temp / edi;
        edx = temp % edi;
        res[ebx].quot = eax;
        res[ebx].rem = edx;
        ebx++;
        goto L;


E:      return;


F:;     eax = ecx[ebx];
        goto G;

H:;     edi = ecx[ebx];
        goto I;
}
```

```
// ASM

.global array_div

array_div:

        pushl %ebx
        pushl %esi
        pushl %edi

        xorl %ebx, %ebx

L:      cmpl 28(%esp), %ebx
        jge E
        movl 16(%esp), %ecx
        movl 20(%esp), %esi

# Prima if
        movl (%esi, %ebx, 4), %eax
        cmpl %eax, (%ecx, %ebx, 4)
        jge F
        movl (%esi, %ebx, 4), %eax
G:
# Seconda if
        movl (%esi, %ebx, 4), %edi
        cmpl %edi, (%ecx, %ebx, 4)
        jl H
        movl (%esi, %ebx, 4), %edi
I:
        movl %eax, %edx
        sarl $31, %edx
        idiv %edi
        movl 24(%esp), %ecx
        movl %eax, (%ecx, %ebx, 8)
        movl %edx, 4(%ecx, %ebx, 8)
        incl %ebx
        jmp L

F:      movl (%ecx, %ebx, 4), %eax
        jmp G

H:      movl (%ecx, %ebx, 4), %edi
        jmp I

E:      popl %edi
        popl %esi
        popl %ebx
        ret
```

Esercitazione 5

Esercizio 2

```c
// Funzione C

#include "e2.h"

int lcm(int x, int y) {
    int greater = y;
    if (x > y)
        greater = x;
    while (1) {
        if ((greater % x == 0) && (greater % y == 0))
            return greater;
        greater++;
    }
}
```

```c
// C Equivalente

#include "e2.h"

int lcm(int x, int y){
        int ecx = y;
        if (x > ecx) ecx = x;
L:      if ((ecx % x) != 0) goto F;
        if ((ecx % y) != 0) goto F;
        return ecx;
F:      ecx++;
        goto L;
}
```

```
// ASM

.global lcm

lcm:

        pushl %ebx

        movl 12(%esp), %ecx
        cmpl %ecx, 8(%esp)
        cmovg 8(%esp), %ecx
L:      movl %ecx, %eax
        movl 8(%esp), %ebx
        movl %eax, %edx
        sarl $31, %edx
        idiv %ebx
        testl %edx, %edx
        jne F

        movl %ecx, %eax
        movl 12(%esp), %ebx
        movl %eax, %edx
        sarl $31, %edx
        idiv %ebx
        testl %edx, %edx
        jne F

        movl %ecx, %eax
        popl %ebx
        ret

F:      incl %ecx
        jmp L
```

Esercizio 3

```c
// Funzione C

#include <string.h>
#include "e3.h"

char charfreq(const char* s) {
    unsigned freq[256];
    memset(freq, 0, 256*sizeof(unsigned));

    while (*s) freq[*s++]++;

    unsigned maxi = 0;
    unsigned maxf = freq[0];
    int i;
    for (i=1; i<256; ++i){
        if (freq[i]>maxf) {
            maxi = i;
            maxf = freq[i];
        }
    }
    return maxi;
}
```

```c
// C Equivalente

#include <string.h>
#include "e3.h"

char charfreq(const char* s){
        unsigned ecx[256];      // freq
        memset(ecx, 0, 1024);
        const char* esi = s;    // s

L1:     if (*esi == 0) goto P;
        ecx[*esi] += 1;
        esi++;
        goto L1;

P:;     unsigned eax = 0;       // maxi
        unsigned edx = ecx[0];  // maxf
        int ebx = 1;
L2:     if (ebx >= 256) goto E;
        if (ecx[ebx] > edx) goto F;
G:      ebx++;
        goto L2;

F:      eax = ebx;
        edx = ecx[ebx];
        goto G;

E:      return eax;
}
```

```
// ASM

.global charfreq

charfreq:

        pushl %ebx
        pushl %esi
        pushl %edi
        subl $1036, %esp            # freq sta nello stack -> devo riservarmi 1024 spazi

        leal 12(%esp), %ecx
        movl %ecx, (%esp)
        movl $0, 4(%esp)
        movl $1024, 8(%esp)
        call memset
        movl %eax, %ecx
        movl 1052(%esp), %esi

L1: movb (%esi), %al
    movzbl %al , %eax
    cmpl $0, %eax
        je P
        incl (%ecx, %eax, 4)
        incl %esi
        jmp L1

P:      xorl %eax, %eax
        xorl %ebx, %ebx
        movl (%ecx, %ebx, 4), %edx
    movl $1, %ebx

L2:     cmpl $256, %ebx
        jge E
        cmpl %edx, (%ecx, %ebx, 4)
        jg F
G:      incl %ebx
        jmp L2

F:      movl %ebx, %eax
        movl (%ecx, %ebx, 4), %edx
        jmp G
E:
        addl $1036, %esp
        popl %edi
        popl %esi
        popl %ebx
        ret
```

Esercizio 4

Scrivere una funzione che, dato un intero x senza segno a 32 bit e un buffer `bin` di 32 caratteri, ottiene la rappresentazione binaria del numero con il byte più significativo per primo.
Gli 0 e 1 nel risultato devono essere rappresentati mediante i caratteri ASCII '0' e '1'.

Esempio: Invocando la funzione con `x = 0x0F0F0F0F`, otteniamo nel buffer di output `bin = "00001111000011110000111100001111"`.

```c
#include <stdio.h>
#include <string.h>
#include <stdlib.h>
#include "e4.h"

void uint2bin(unsigned x, char bin[32]){
    int i;
    for (i = 0; i < 32; i++){
        bin[i] = '0' + (x & 1);
        x = x >> 1;
        }
    return;
}

// Faccio la and bit a bit di x -> metto 1 se bit = 1
//                              -> metto 0 se bit = 0
```

Preparazione 6

Esercizio 1
La funzione verifica se il suo parametro c è un carattere di punteggiatura preso da una lista limitata. Scriverne una versione ottimizzata.

```c
// Funzione non ottimizzata

#include "e1.h"

int is_punctuation(char c) {
    return  c=='.'  || c=='-' || c==';' || c==':' || c=='\'' ||
            c=='_'  || c=='(' || c==')' || c=='/' || c=='|'  ||
            c=='\\' || c=='[' || c==']' || c=='<' || c=='>'  ||
            c=='|'  || c=='?' || c=='!' || c==',';
}
```

Soluzione
Le tecniche di ottimizzazione applicate sono:

- *Cortocircuitazione*
- *Compile-time inizialization (o pre-computazione)*

```c
// Funzione ottimizzata

#include "e1.h"

int lookup_table[256] = {
    0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0,
    0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0,
    0, 1, 0, 0, 0, 0, 0, 1, 1, 1, 0, 0, 1, 1, 1, 1,
    0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 1, 1, 1, 0, 1, 1,
    0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0,
    0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 1, 1, 1, 0, 1,
    0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0,
    0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 1, 0, 0, 0,
    0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0,
    0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0,
    0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0,
    0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0,
    0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0,
    0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0,
    0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0,
    0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0,
};

int is_punctuation(char c) {

    return lookup_table[(unsigned char)c];
}
```

Esercitazione 6

Header -> Preparazione 3

Esercizio 1

```c
// Funzione C

#include <stdlib.h>
#include "e1.h"

void list_concat(node_t **l1, node_t *l2) {
    node_t *p = *l1;
    if (p==NULL) *l1 = l2;
    else {
        while (p->next!=NULL) p = p->next;
        p -> next = l2;
    }
}
// C Equivalente

#include <stdlib.h>
#include "e1.h"

void list_concat(node_t **l1, node_t *l2){
    node_t** eax = l1;
    node_t* ecx = *eax;
    if (ecx == NULL) goto E1;
L:  if ((*ecx).next == NULL) goto E2;
    ecx = (*ecx).next;
    goto L;

E2: (*ecx).next = l2;
    return;

E1: *eax = l2;
    return;
    }
// ASM

.global list_concat
list_concat:
    movl 4(%esp), %eax
    movl (%eax), %ecx
    cmpl $0, %ecx
    je E1
L:  cmpl $0, 4(%ecx)
    je E2
    movl 4(%ecx), %ecx
    jmp L

E1: movl 8(%esp), %edx
    movl %edx, (%eax)
    ret

E2: movl 8(%esp), %edx
    movl %edx, 4(%ecx)
    ret
```

Esercizio 2

Un tipico formato per rappresentare immagini digitali è la matrice `row-major`, dove le righe sono disposte consecutivamente in memoria.
Per matrici a toni di grigio a 8 bit, ogni cella dell'array è compresa tra a (nero) e 255 (bianco). In totale, per un'immagine di altezza `h` e ampiezza `w`, l'array contiene `w*h` celle. La cella `(0,0)` è collocata nell'angolo superiore sinistro dell'immagine. Il pixel di coordinate `(i,j)`, dove `i` è la coordinata verticale e `j` quella orizzontale, risiede nella cella di indice `v[i*w+j]` dell'array.

Scrivere una funzione che applica a un'immagine di input a 256 toni di grigio `(unsigned char)` un classico filtro che consente di sfocare l'immagine. I parametri sono:

- `in`: array della matrice di input da sfocare
- `out`: array della matrice di output sfocata
- `w`: ampiezza delle immagini di input e di output (indici `j` in `[0,w]`)
- `h`: altezza delle immagini di input e di output (indici `i` in `[0,h]`)

Il filtro da applicare è un semplice procedimento di convoluzione: ogni pixel di coordinate `(i,j)` dell'output sarà calcolato come la media aritmetica dei 25 valori in input nella finestra 5x5 centrata in `(i,j)`.
I pixel di output vicino ai bordi, la cui finestra 5x5 uscirebbe dai bordi dell'immagine di input, prendono semplicemente il valore del corrispondenti pixel di input.

```c
#include "e2.h"

void blur5(unsigned char* in, unsigned char* out, int w, int h){
    int i,j,k,l;
    int somma;
    for (i = 0; i < h; i++){
        for (j = 0; j < w; j++){
            if(i >= 2 && i < h-2 && j >= 2 && j < w-2){
                somma = 0;
                for (k = i-2; k < i+3; k++){
                    for (l = j-2; l < j+3; l++){
                        somma += in[k*w+l];
                        }
                    }
                out[i*w+j] = somma/25;
                }
            else{
                out[i*w+j] = in[i*w+j];
                }
            }
        }
    }
```

Preparazione 7

Esercizio 1
Scrivere nel file una funzione C che cerca se x appartiene all'array v di dimensione n.
La soluzione deve usare almeno due processi distinti che effettuano la ricerca in sottoparti distinte di v in modo indipendente l'uno dall'altro.

Suggerimento: fare in modo che il processo restituisca al genitore come stato di exit 1 se x è presente nella porzione esplorata dal processo, e 0 altrimenti.

```c
#include "e1.h"
#include <stdio.h>
#include <stdlib.h>
#include <unistd.h>
#include <sys/wait.h>
#include "e1.h"

#define NUM 4

int aux(int* v, unsigned n, int x){
    int i;
    for (i = 0; i < n; i++)
        if (v[i] == x) return 1;
    return 0;
    }


int par_find(int* v, unsigned n, int x){
    int i;
    int totale;

    for (i = 0; i < NUM; i++){
        pid_t pid = fork();
        if (pid == -1){
            perror("Errore nella fork:");
            exit(1);
            }
        if (pid == 0){
            int find = aux(v+i*n/NUM, n/NUM, x);
            _exit(find);
            }
        }

    for (i = 0; i < NUM; i++){
        int status;
        wait(&status);
        if(WIFEXITED(status))
            totale = totale || WEXITSTATUS(status);
    }

    return totale;
}
```

Esercitazione 7

Header

```c
#ifndef __GET_CMD_LINE__
#define __GET_CMD_LINE__

#define MAX_LINE    1024
#define MAX_TOKENS  64

void get_cmd_line(char* argv[MAX_TOKENS]);

#endif
```

Esercizio 1

Il canale `stdin` (di tipo `FILE*`) modella in C la sorgente di caratteri ASCII che proviene come input, salvo diversamente specificato, dal terminale.
Si chiede di scrivere una funzione che legge la prossima linea di testo da `stdin`, estrae ciascun token (sequenza consecutiva di caratteri, esclusi spazi, tab `\t` e ritorni a capo `\n`) e produce un array di al più `n<=MAX_TOKENS=64` stringhe come segue:

- Le stringhe prodotte `argv[0]`…`argv[n-1]` devono essere allocate dinamicamente con `malloc`.
- La stringa in ultima posizione `argv[n]` deve essere `NULL`, fungendo da "terminatore".

Assumere che la linea di testo letta da `stdin` contenga al più 1024 caratteri compreso il ritorno a capo `\n`.

Suggerimento: Basandosi sul comando `man` usare:

- La funzione `fgets` per leggere una linea da `stdio` di al più `MAX_LINE=1024` caratteri terminata dal ritorno a capo `\n`.
- La funzione `strtok` per tokenizzare la linea una volta letta da `stdio`.

```c
#include <fcntl.h>
#include <unistd.h>
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include "e1.h"

void get_cmd_line(char* argv[MAX_TOKENS]){
    char* stringa = (char*)malloc(sizeof(char) * MAX_LINE);
    char* tokens = (char*)malloc(sizeof(char) * MAX_LINE);
    const char* tok = " \n\t";
    fgets(stringa, MAX_LINE, stdin);
    tokens = strtok(stringa, tok);
    int i = 0;
    while (i <= MAX_TOKENS && tokens){
        argv[i] = (char*)malloc(sizeof(char) * MAX_LINE);
        strcpy(argv[i], tokens);
        tokens = strtok(NULL, tok);
        i++;
        }

    argv[i] = NULL;
}
```

Esercizio 2

Una `shell` è un programma che chiede all'utente di eseguire altri programmi sotto forma di nuovi processi, passandogli eventuali argomenti specificati dall'utente.
Una `shell` fornisce normalmente un `prompt` (es. $, >>, ecc.) che segnala all'utente che la `shell` è in attesa di ricevere comandi.

Si chiede di scrivere una semplice `shell` sotto forma di una funzione che prende come parametro la stringa di `prompt` e si comporta come segue:

- Stampa il `prompt`.
- Attende che l'utente inserisca in `stdin` un comando seguito dai suoi eventuali argomenti.
  Per ottenere comando e argomenti da `stdin` usare il risultato dell'esercizio 2.
- Se il comando è vuoto (`NULL`) tornare al punto 1.
- Se il comando è `quit` terminare con successo la `shell`.
- Creare con `fork` un nuovo processo che esegua il comando con gli argomenti dati usando `execvp`.
- Se il comando si riferisce a un programma inesistente riportare l'errore `unknown command` seguito dal nome del comando e tornare al punto 1.
- Attendere con `wait` la terminazione del processo e tornare al punto 1.

Il risultato di **ogni** system call **deve** essere controllato e in caso di errore segnalato con `perror` e terminazione `EXIT_FAILURE`.

```c
#include <fcntl.h>
#include <unistd.h>
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include "e1.h"

void get_cmd_line(char* argv[MAX_TOKENS]){
    char* stringa = (char*)malloc(sizeof(char) * MAX_LINE);
    char* tokens = (char*)malloc(sizeof(char) * MAX_LINE);
    const char* tok = " \n\t";
    fgets(stringa, MAX_LINE, stdin);
    tokens = strtok(stringa, tok);
    int i = 0;
    while (i <= MAX_TOKENS && tokens){
        argv[i] = (char*)malloc(sizeof(char) * MAX_LINE);
        strcpy(argv[i], tokens);
        tokens = strtok(NULL, tok);
        i++;
        }

    argv[i] = NULL;
}


int do_shell(const char* prompt){
    int i = 1;
    char* argv[MAX_TOKENS];
    printf("Inserire exit per uscire\n");
    while(i){
        printf("%s", prompt);
        get_cmd_line(argv);
        if((strcmp(argv[0], "exit")) == 0)
            exit(0);

        pid_t pid = fork();

        if (pid == -1){
            perror("Errore nella fork: ");
            exit(1);
            }

        if (pid == 0){

            int exec = execvp(argv[0], argv);
            if (exec != 0){
                perror("Errore nella execvp: ");
                exit(1);
                }
            }


            wait(NULL);

        }

}
```

Preparazione 8

Esercizio 1

Si vuole scrivere un programma che calcola la somma dei valori senza segno a 32 bit contenuti in un file, ignorando eventuali byte finali resto della divisione per 4, se la lunghezza del file in byte non è divisibile per 4.
La lettura deve avvenire 4 byte alla volta.
La funzione che prende il nome di un file e l'indirizzo di una variabile dove scrivere il numero calcolato.

```c
#include <unistd.h> // read, write, close
#include <fcntl.h>  // open
#include <stdlib.h> // exit
#include <stdio.h>  // perror

#include "e1.h"


int sum(const char* filename, unsigned long *psum) {

    unsigned long somma = 0;
    unsigned buffer;
    ssize_t letti;

    int file = open(filename, O_RDONLY);
    if (file == -1){
        perror("Errore apertura file: ");
        exit(1);
        }

    while(1){

        letti = read(file, &buffer, sizeof(buffer));

        if (letti == -1){
            perror("Errore nella read: ");
            exit(1);
            }

        if (letti == 4)
            somma += buffer;

        if (letti == 0)
            break;
    }

    file = close(file);
    if (file == -1){
        perror("Errore chiusura file: ");
        exit(1);
        }

    *psum = somma;

    return 0;
}
```

Esercizio 2
Lo scopo dell'esercizio è quello di scrivere file binari con la seguente struttura composti da numeri pseudo-casuali:
Il magic number deve essere 0xEFBEADDE per tutti i file generati con questo programma.

DEFINIZIONE: I *magic number* sono codici associati ai tipi di file, specialmente se binari, per identificarli univocamente al momento dell'apertura, e sono disposti all'inizio del file.

Si richiede di scrivere una funzione che genera il file a partire dai dati forniti da riga di comando, secondo con i seguenti parametri:

- `size`: numero di valori casuali registrati nel file.
- `seed`: seme del generatore pseudo-casuale.
- `mod`: limite superiore per i valori casuali, in valore assoluto.
- `filename`: percorso assoluto o relativo del file da generare.

```c
#include <unistd.h>
#include <stdio.h>
#include <stdlib.h>
#include <sys/stat.h>
#include <fcntl.h>
#include "../e2.h"

int make_rnd_file(unsigned size, unsigned seed, unsigned mod, char *filename) {

    int res, data, fd, i, magic = MAGIC_NUMBER;

    // set pseudo-random generator seed
    srand(seed);

    // open file
    mode_t mode = S_IRUSR | S_IWUSR | S_IRGRP | S_IROTH;
    fd = open(filename, O_WRONLY | O_TRUNC | O_CREAT, mode);
    check_perror(fd, "open");

    // write file magic code
    res = write(fd, &magic, sizeof(magic));
    check_perror(res, "write");

    // write size of the file in terms of random ints
    res = write(fd, &size, sizeof(size));
    check_perror(res, "write");

    // write random data
    for (i=0; i < size; ++i){
        data = rand() % mod;
        res = write(fd, &data, sizeof(data));
        check_perror(res, "write");
    }

    res = close(fd);
    check_perror(res, "write");

    return EXIT_SUCCESS;
}
```

Esercizio 4

```c
#include <unistd.h> // read, write, close
#include <fcntl.h>  // open
#include <stdlib.h> // exit
#include <stdio.h>  // perror
#include "e4.h"

// Dati i percorsi di due file f1 e f2 restituisce zero se i file sono uguali
// un valore maggiore di zero se i file sono diversi, e -1 in caso di errore.

int res = 0;
void check_error(char* err_msg){
    perror(err_msg);
    res = -1;
    exit(1);
}

int file_eq(char* f1, char* f2) {
    char* buf1 = (char*)malloc(sizeof(char));
    char* buf2 = (char*)malloc(sizeof(char));
    ssize_t read1;
    ssize_t read2;

    int fd1 = open(f1, O_RDONLY);
        check_error("Errore apertura file1: ");

    int fd2 = open(f2, O_RDONLY);
    if (fd2 == -1){
        check_error("Errore apertura file2: ");

    while(1){

        read1 = read(fd1, buf1, sizeof(buf1));
        if (read1 == -1)
            check_error("Errore nella read file1: ");

        read2 = read(fd2, buf2, sizeof(buf2));
        if (read2 == -1)
            check_error("Errore nella read file2: ");

        if ((memcmp(buf1, buf2, read2)) || read1 != read2)
            res = 1;

        if (read1 == 0 && read2 == 0)
            break;
        }

    fd1 = close(fd1);
    if (fd1 == -1)
        check_error("Errore nella close file1: ");

    fd2 = close(fd2);
    if (fd2 == -1)
        check_error("Errore nella close file2: ");

    free(buf1);
    free(buf2);
    return res;
}
```

Esercitazione 8

Header -> Preparazione 3

Esercizio 1

```c
// Funzione C

#include "e1.h"
#include <stdlib.h>

int list_equal_array(const node_t *p, short *buf, int n) {
    int i;
    for (i = 0; i < n; i++) {
        if (p == NULL || p->elem != buf[i]) return 0;
        p = p->next;
    }
    return p == NULL;
}
```

```c
// C Equivalente

#include <stdlib.h>
#include "e1.h"

int list_equal_array(const node_t* p, short* buf, int n){
    const node_t* eax = p;
    int ecx = 0;
    short* edx = buf;
L:  if (ecx >= n) goto E;
    if (eax == NULL) goto E0;
    if ((*eax).elem != edx[ecx]) goto E0;
    eax = (*eax).next;
    ecx++;
    goto L;

E:  if(eax == NULL) goto E1;
E0: return 0;

E1: return 1;
}
```

```
// ASM

.global list_equal_array

list_equal_array:

    pushl %ebx

    movl 8(%esp), %eax
    xorl %ecx, %ecx
    movl 12(%esp), %edx

L:  cmpl 16(%esp), %ecx
    jge E
    cmpl $0, %eax
    je E0
    movl (%edx, %ecx, 2), %ebx
    cmpl %ebx, (%eax)
    je E0
    movl 4(%eax), %eax
    incl %ecx
    jmp L

E:  cmpl $0, %eax
    je E1

E0: xorl %eax, %eax
    popl %ebx
    ret

E1: movl $1, %eax
    popl %ebx
    ret
```

Esercizio 2
Scrivere una funzione che realizza la copia di un file di dimensione arbitraria con nome file sorgente `src` e destinazione `dst`. La funzione deve restituire `EXIT_FAILURE` in caso di errore ed `EXIT_SUCCESS` altrimenti.

Suggerimento: allocare dinamicamente un buffer da usare per il travaso di dati dal file sorgente al file destinazione. Provare con dimensioni diverse scegliendo la minima ragionevole per mantenere le prestazioni.

```c
#include <unistd.h> // read, write, close
#include <fcntl.h>  // open
#include <stdlib.h> // exit
#include <stdio.h>  // perror
#include "e2.h"
#define BUF_SIZE 32

void check_error(char* err_msg){
    perror(err_msg);
    res = -1;
    exit(1);
}

int copy(const char* src, const char* dst){
    char* buffer = (char*)malloc(sizeof(char) * BUF_SIZE);
    ssize_t letto;
    ssize_t scritto;
    int f1 = open(src, O_RDONLY);
    if (f1 == -1)
        check_error("Errore apertura file src: ");

    int f2 = open(dst, O_WRONLY | O_CREAT | O_TRUNC, 0644);
    if (f2 == -1)
        check_error("Errore apertura file dest: ");

    while(1){
        letto = read(f1, buffer, sizeof(buffer));
        if (letto == -1)
            check_error("Errore in lettura: ");

        if (letto == 0)
            break;

        scritto = write(f2, buffer, letto);
        if (scritto == -1)
            check_error("Errore nella scrittura: ");
        }

    f1 = close(f1);
    if (f1 == -1)
        check_error("Errore nella close: ");

    f2 = close(f2);
    if (f2 == -1){
        check_error("Errore nella close: ");

    free(buffer);
    return 0;
    }
```

Preparazione 9

Esercizio 1

Si vuole scrivere un programma che prende come parametro il numero N figli, tale che il figlio i-esimo dorme i secondi (con i=1, …, N), strutturato nel seguente modo:

- Per ogni figlio creato, il programma stampa: "`- creato figlio <pid>`".
- Per ogni figlio terminato, il gestore del segnale stampa: "`terminato figlio <pid>`".
- La terminazione dei figli è catturata tramite il segnale `SIGCHLD`: il padre resta in attesa perenne mentre il gestore del segnale fa una `wait`, quando l'ultimo processo termina allora il gestore del segnale manda in segnale `SIGTERM` a sé stesso per terminare anche il genitore.

```c
#include<unistd.h>
#include<stdio.h>
#include<stdlib.h>
#include<sys/wait.h>

int N;

void handler(int signo){
    int status;
    pid_t pid = wait(&status);
    printf("Ricevuto segnale SIGCHILD dal figlio <%d> \n", pid);
    N--;
    if (N == 0)
        kill(getpid(), SIGTERM);
}

void do_work(int n){
    N = n;
    struct sigaction act ={0};
    act.sa_handler = handler;
    if (sigaction(SIGCHLD, &act, NULL) == -1){
        perror("Errore nella sigaction");
        exit(1);
            }

    printf("Processo padre <%d> \n", getpid());
    pid_t pid;
    int i;
    for (i = 0; i < n; i++){
        pid = fork();
        if (pid == -1){
            perror("Errore nella fork");
            exit(1);
            }
        if (pid == 0){
            printf("Creato figlio <%d> \n", getpid());
            sleep(i);
            exit(0);
            }
        }

    while(1)
        pause();

    printf("Processo padre termina\n");
    return;
}
```

Esercitazione 9

Esercizio 1

```c
// Funzione C

#include "e1.h"

int matsum(int** m, int n){
    int i, j, s = 0;
    for (i=0; i<n; ++i)
        for (j=0; j<n; ++j)
            s += m[i][j];
    return s;
}
```

```c
// C Equivalente

#include "e1.h"

int matsum(int** m, int n){
    int eax = 0;    // totale
    int ebx;        // i
    int ecx;        // j
    int** edx = m;

    ebx = 0;
L1: if(ebx >= n) goto E;
    ecx = 0;
L2: if (ecx >= n) goto F;
    eax += edx[ebx][ecx];
    ecx++;
    goto L2;

F:  ebx++;
    goto L1;

E:  return eax;
    }
```

```
// ASM

.global matsum

matsum:

    pushl %ebx
    pushl %esi
    pushl %edi

    xorl %eax, %eax
    xorl %ebx, %ebx
    movl 16(%esp), %edx
L1: cmpl 20(%esp), %ebx
    jge E
    xorl %ecx, %ecx
L2: cmpl 20(%esp), %ecx
    jge F
    movl (%edx, %ebx, 4), %esi
    movl (%esi, %ecx, 4), %edi
    addl %edi, %eax
    incl %ecx
    jmp L2
F:  incl %ebx
    jmp L1

E:  popl %edi
    popl %esi
    popl %ebx
    ret
```

Esercizio 2

Si vogliono usare i segnali per creare un indicatore di progresso che implementa un semplice algoritmo di ordinamento a bolle. L'indicatore di progresso è la percentuale di n coperta da i, vale a dire 100.0*i/n.

Suggerimento: rendere la variabile `i` globale (dichiarata fuori dalla funzione) e tenere in un'altra variabile globale `max` il valore di n. In questo modo è possibile accedervi da un `handler` di un segnale che può stampare il rapporto tra `i` e `max`.

```c
#include <signal.h>
#include <stdio.h>
#include <stdlib.h>
#include <unistd.h>

#include "e2.h"

int I, N;

void handler(int signo) {
    float percentuale = (100.0*I)/N;
    printf("%f %% \n", percentuale);
}

static void do_sort(int *v, int n) {
    int i, j;
    for (i=0; i<n; ++i){
        I = i;
        for (j=1; j<n; ++j)
            if (v[j-1] > v[j]) {
                int tmp = v[j-1];
                v[j-1] = v[j];
                v[j] = tmp;
            }
        }
}

void sort(int *v, int n) {

    N = n;

    struct sigaction act = { 0 };
    act.sa_handler = handler;
    if (sigaction(SIGALRM, &act, NULL) == -1){
        perror("sigaction");
        exit(EXIT_FAILURE);
    }

    ualarm(750000U,750000U);
    do_sort(v, n);
}
```

Esercitazione 10

Esercizio 1

```c
// Funzione C

#include "e1.h"

int is_prefix(const char* s1, const char* s2){
    while (*s1 && *s1++ == *s2++);
    return *s1 == 0;
}
```

```c
// C Equivalente

#include "e1.h"

int is_prefix(const char* s1, const char* s2){
    const char* eax = s1;
    const char* ecx = s2;
    char edx;
L:  if (*eax == 0) goto E;
    edx = *ecx;
    if (*eax != edx) goto E;
    eax++;
    ecx++;
    goto L;

E:  if(*eax == 0) goto E1;
    goto E0;

E1: return 1;
E0: return 0;
    }
```

```
// ASM

.global is_prefix

is_prefix:

    movl 4(%esp), %eax
    movl 8(%esp), %ecx
L:  cmpb $0, (%eax)
    je E
    movb (%ecx), %dl
    cmpb %dl, (%eax)
    jne E
    incl %eax
    incl %ecx
    jmp L

E:  cmpb $0, (%eax)
    je E1
    xorl %eax, %eax
    ret

E1: movl $1, %eax
    ret
```

Esercizio 2

Implementare l'operazione di deallocazione di un semplice allocatore di memoria.
Data la funzione `mymalloc` fornita, si richiede di scrivere la funzione `myfree` che prende il blocco deallocato e lo inserisce in testa alla lista di blocchi liberi puntata dalla variabile globale `free_list`.
I blocchi liberi dell'allocatore hanno una `header` di 4 byte che contiene la dimensione del blocco, seguita da un campo `next` come definito nel file **header**:

```c
#ifndef __E2__
#define __E2__

#pragma pack(1)
typedef struct header_t {
    unsigned size;
    struct header_t* next;
} header_t;

void* mymalloc(size_t m);
void myfree(void* p);

#endif
```

Nota Bene: La direttiva del compilatore `#pragma pack(1)` evita l'inserimento di `padding` tra il campo `size` e il campo `next`, che quindi sono contigui in memoria.

```c
#include <unistd.h>
#include <stdlib.h>
#include "e2.h"

header_t* free_list = NULL;

void* mymalloc(size_t m) {
    header_t *p, *q = NULL;
    m = ((m+3)/4)*4;                // arrotonda al più piccolo multiplo di 4 maggiore o uguale a m
    if (m < 8) m = 8;               // garantisce spazio per il puntatore next
    for (p=free_list; p != NULL; q = p, p = p->next)    // cerca blocco libero first-fit
        if (p->size >= m) break;
    if (p == NULL) p = sbrk(4 + m); // nessun blocco libero, si espande l'heap
    else                            // toglie nodo dalla lista dei blocchi liberi
        if (q == NULL) free_list = free_list->next;
        else q->next = q->next->next;
    ((header_t*)p)->size = m;       // size misura il blocco a meno del campo size stesso
    return (char*)p+4;
}

void myfree(void* p) {
    header_t* q = (header_t*)((char*)p-4);
    q->next = free_list;
    free_list = q;
}
```

Esercitazione 11

Esercizio 1

```c
// Funzione C

#include "e1.h"

int sub_array(const short *a, unsigned na, const short *b, unsigned nb) {
    int i;
    for (i=0; i+na <= nb; ++i) {
        int res;
        is_prefix(a, b+i, na, &res);
        if (res) return 1;
    }
    return 0;
}
```

```c
// C Equivalente

#include "e1.h"

int sub_array(const short* a, unsigned na, const short* b, unsigned nb){
    int esi = 0;            // i
    unsigned ebx = na;      // na
    unsigned ecx = nb;      // nb
    unsigned edx;           // i+na
L:  edx = esi + ebx;
    if (edx > ecx) goto E0;
    int res;
    is_prefix(a, b+esi, ebx, &res);
    if (res != 0) goto E1;
    esi++;
    goto L;

E0: return 0;

E1: return 1;
    }
```

Nota Bene: Attenzione all'aritmetica dei puntatori e a tenere `res` in `stack`.

```
// ASM

.global sub_array

sub_array:

    pushl %ebx
    pushl %esi
    pushl %edi
    subl $20, %esp

    xorl %esi, %esi

L:  movl 40(%esp), %ebx     # na
    movl 48(%esp), %ecx     # nb
    movl %esi, %edx
    addl %ebx, %edx
    cmpl %ecx, %edx
    jg E0
    movl 36(%esp), %edi     # a
    movl %edi, (%esp)
    movl 44(%esp), %edi     # b
    leal (%edi, %esi, 2), %edi
    movl %edi, 4(%esp)
    movl %ebx, 8(%esp)
    leal 16(%esp), %edi
    movl %edi, 12(%esp)
    call is_prefix
    cmpl $0, 16(%esp)
    jne E1
    incl %esi
    jmp L

E0: xorl %eax, %eax
    jmp E

E1: movl $1, %eax
E:
    addl $20, %esp
    popl %edi
    popl %esi
    popl %ebx
    ret
```

Esercizio 2

Scrivere una versione **ottimizzata** della funzione fornita, che sia semanticamente equivalente, e che faccia un migliore uso delle cache.

```
// Versione non ottimizzata

#include "e2.h"

#define STRIDE 64

long f(const short *v, unsigned n){
    long x = 0;
    unsigned i, j;
    for (i=0; i<STRIDE; ++i)
        for (j=0; j<n; j+=STRIDE) x += v[i+j];
    return x;
}
```

Spiegazione e soluzione

La funzione `f` parte da indice `i` pari a 0 e fa scorrere l'indice `j` di 64 posizioni alla volta (a partire da 0).
Così facendo, vado ad estrarre inizialmente solo gli elementi multipli di 64+0(=`i`).
Poi incrementa `i`, e vado a prendere gli elementi multipli di 64+1(=`i`).
Poi incrementa `i`, e vado a prendere gli elementi multipli di 64+2(=`i`).
E così via…
Le prime posizioni, quelle a partire da indice 0 fino a 63, sono coperte dal primo ciclo del for più interno quando `j=0`.

```
// Versione ottimizzata

#include "e2.h"

long f_opt(const short *v, unsigned n){
    long x = 0;
    unsigned i;
    for (i = 0; i < n; i++)
        x+=v[i];
    return x;
}
```

Preparazione 12

Esercizio 1

```c
// Funzione C

#include "e1.h"

void selection_sort(short *v, int n) {
    int i,j, min;
    for (i=0; i<n-1; ++i) {
        min = i;
        for (j=i+1; j<n; ++j)
            if (v[j]<v[min]) min = j;
        swap(v+i, v+min);
    }
}
```

```c
// C Equivalente

#include "e1.h"

void selection_sort(short* v, int n){
    int eax = 0;          // i
    int ebx;              // j
    int ecx;              // min
    int edx = n;
    int edi = edx-1;
    short* esi = v;
L1: if (eax >= edi) goto E;
    ecx = eax;
    ebx = eax+1;
L2: if (ebx >= edx) goto F;
    if (esi[ebx] < esi[ecx]) ecx = ebx;
    ebx++;
    goto L2;

F:  swap(esi+eax, esi+ecx);
    eax++;
    goto L1;

E:  return;
    }
```

```
// ASM

.global selection_sort

selection_sort:

    pushl %ebx
    pushl %edi
    pushl %esi
    subl $8, %esp

    xorl %eax, %eax

L1:
    movl 24(%esp), %esi

    movl 28(%esp), %edx
    movl %edx, %edi
    decl %edi
    cmpl %edi, %eax
    jge E
    movl %eax, %ecx
    movl %eax, %ebx
    incl %ebx

L2:
    cmpl %edx, %ebx
    jge F
    movw (%esi, %ecx, 2), %di
    cmpw %di, (%esi, %ebx, 2)
    cmovll %ebx, %ecx
    incl %ebx
    jmp L2

F:
    leal (%esi, %eax, 2), %edx
    movl %edx, (%esp)
    leal (%esi, %ecx, 2), %edx
    movl %edx, 4(%esp)
    call swap
    incl %eax
    jmp L1

E:
    addl $8, %esp
    popl %esi
    popl %edi
    popl %ebx
    ret
```

Esercizio 2

Scrivere una funzione che prende come parametri:

- Il pathname di un eseguibile `file`.
- Un puntatore a una funzione callback `get_arg` fornita dall'utente.
- Un puntatore a dei dati `data` da fornire alla funzione callback
- Un numero di ripetizioni n.

La funzione esegue `file` per n volte, misurando il tempo in secondi (`double`) richiesto dall'esecuzione e restituisce la media dei tempi richiesti sulle n esecuzioni.
Ad ogni esecuzione, la funzione chiama la callback per ottenere gli argomenti `argv` da passare al processo in modo da rendere possibile comportamenti diversi del processo ad ogni ripetizione.
La callback deve ricevere il puntatore data preso come parametro da `proc_gettime` e il numero della ripetizione i con i compreso tra `0` e `n-1`.

Suggerimento: usare `clock_gettime` per misurare il tempo.

```c
#include "e2.h"
#include <stdio.h>
#include <stdlib.h>
#include <assert.h>
#include <time.h>  // clock_gettime
#include <unistd.h>
#include <sys/wait.h>
#include <fcntl.h>

double get_real_time() {
        struct timespec ts;
        clock_gettime(CLOCK_MONOTONIC, &ts);
    return ts.tv_sec + ts.tv_nsec*1E-09;
}


double proc_gettime(const char* file, char** (*get_argv)(int i, void* data), void *data, int n){

    int i = 0;
    double start, elapsed = 0.0;

    for (i = 0; i < n; i++){
        pid_t pid = fork();
        if (pid == -1){
            perror("Errore nella fork: ");
            exit(1);
            }
        if (pid == 0){
            int exec = execvp(file, get_argv(i, data));
            if (exec != 0){
                perror("Errore nella execvp: ");
                exit(1);
                }
            }

        start = get_real_time();
        wait(NULL);
        elapsed += get_real_time() - start;

    }

    return elapsed/n;

}
```

Esercizio 3
Si vuole scrivere una funzione che misura il tempo medio in millisecondi per accesso a disco in lettura e scrittura in modo casuale e in modo sequenziale.

```c
#ifndef __E3__
#define __E3__

typedef struct {
    unsigned file_size; // [xxxx....]
    double seq_read_t;  // [xxxxxxxx]
    double seq_write_t; // [xxxxxxxx]
    double rnd_read_t;  // [xxxxxxxx]
    double rnd_write_t; // [xxxxxxxx]
} time_rec_t;           // sizeof(struct) = 40

void file_access_time(const char* file, unsigned trials, time_rec_t *t);

#endif
```

La funzione ha il seguente prototipo:

- `file` è il file da accedere (sovrascrivendolo).
- `trials` è il numero di accessi casuali da effettuare sia per le letture che per le scritture.
- `t` è una struttura definita come segue:
  - `file_size` è la dimensione in byte del file
  - `seq_read_t` è il tempo medio in millisecondi per lettura sequenziale di 4 byte
  - `seq_write_t` è il tempo medio in millisecondi per scrittura sequenziale di 4
  - `byternd_read_t` è il tempo medio in millisecondi per lettura casuale di 4 byte
  - `rnd_write_t` è il tempo medio in millisecondi per scrittura casuale di 4 byte

Tutti i campi della struttura devono essere riempiti dalla funzione.
Le letture e le scritture sequenziali devono scorrere l'intero file.
L'eseguibile può essere compilato con make ed eseguito con ./e3 file trials dove file e trials sono definiti come sopra. È possibile generare un file di prova di file.dat da 390 MB con make file. Per ripulire la directory alla fine dell'esperimento usare make clean.

```c
#include "../e3.h"
#include <sys/types.h>
#include <sys/stat.h>
#include <fcntl.h>
#include <stdio.h>
#include <stdlib.h>
#include <time.h>
#include <sys/types.h>
#include <unistd.h>

double get_real_time_msec() {
    struct timespec tp;
    clock_gettime(CLOCK_MONOTONIC, &tp);
    return tp.tv_sec*1E3 + tp.tv_nsec*1E-6;
}

void check_error(int err, char* msg) {
    if (err != -1) return;
    perror(msg);
    exit(EXIT_FAILURE);
}
```

```c
void file_access_time(const char* file, unsigned trials, time_rec_t *t) {
    unsigned i, val, cnt;
    double start, elapsed;
    off_t off;
    int res, fd = open(file, O_RDWR);
    check_error(fd, "open");

    t->file_size = lseek(fd, 0, SEEK_END);              // get file size
    check_error(t->file_size, "lseek");

    printf("File size: %f MB\n", t->file_size/(1024.0*1024.0));

    printf("Performing %u random reads...\n", trials);  // random read
    start = get_real_time_msec();
    for (i=0; i<trials; ++i) {
        off = rand() % t->file_size;
        off = lseek(fd, off, SEEK_SET);
        check_error(off, "lseek");
        res = read(fd, &val, sizeof(val));
        check_error(res, "read");
    }
    t->rnd_read_t = (get_real_time_msec() - start)/trials;

    printf("Performing %u random writes...\n", trials); // random write
    start = get_real_time_msec();
    for (i=0; i<trials; ++i) {
        off = rand() % t->file_size;
        off = lseek(fd, off, SEEK_SET);
        check_error(off, "lseek");
        res = write(fd, &val, sizeof(val));
        check_error(res, "read");
    }
    t->rnd_write_t = (get_real_time_msec() - start)/trials;

    printf("Reading the file sequentially...\n");       // sequential read
    off = lseek(fd, 0, SEEK_SET);
    check_error(off, "lseek");
    cnt = 0;
    start = get_real_time_msec();
    for (;;) {
        cnt++;
        res = read(fd, &val, sizeof(val));
        check_error(res, "read");
        if (res == 0) break;
    }
    t->seq_read_t = (get_real_time_msec() - start)/cnt;

    printf("Writing the file sequentially...\n");       // sequential write
    off = lseek(fd, 0, SEEK_SET);
    check_error(off, "lseek");
    start = get_real_time_msec();
    for (i=0; i<cnt; ++i) {
        res = write(fd, &val, sizeof(val));
        check_error(res, "read");
        if (res == 0) break;
    }
    t->seq_write_t = (get_real_time_msec() - start)/cnt;

    res = close(fd);
    check_error(res, "close");
}
```

Esercitazione 12

Esercizio 1

```c
// Funzione C

#include "e1.h"

void bubble_sort(short *v, unsigned n) {
    unsigned i, again;
    do {
        again = 0;
        for (i=1; i<n; ++i)
            if (v[i-1] > v[i]) {
                swap(&v[i-1], &v[i]);
                again = 1;
            }
    } while(again);
}
```

```c
// Equivalente C

#include "e1.h"

void bubble_sort(short* v, unsigned n){
    unsigned eax;   // i
    unsigned ebx;   // again
    short* ecx = v;
    unsigned edx;
    unsigned edi;

L1: ebx = 0;
    edx = 1;

L2: edi = edx-1;
    if (edx >= n) goto F;
    short esi = ecx[edx];
    if (ecx[edi] <= esi) goto G;
    swap(&ecx[edi], &ecx[edx]);
    ebx = 1;

G:  edx++;
    goto L2;

F:  if(ebx != 0) goto L1;
    return;
    }
```

```
// ASM

.global bubble_sort

bubble_sort:

    pushl %ebx
    pushl %esi
    pushl %edi
    subl $8, %esp

    movl 24(%esp), %ecx

L1: xorl %ebx, %ebx
    movl $1, %esi
    movl $0, %edi

L2: cmpl 28(%esp), %esi
    jae F
    movw (%ecx, %esi, 2), %dx
    cmpw %dx, (%ecx, %edi, 2)
    jle G
    leal (%ecx, %edi, 2), %eax
    movl %eax, (%esp)
    leal (%ecx, %esi, 2), %eax
    movl %eax, 4(%esp)
    call swap
    movl $1, %ebx

G:  incl %esi
    incl %edi
    jmp L2

F:  cmpl $0, %ebx
    jne L1

    addl $8, %esp
    popl %edi
    popl %esi
    popl %ebx
    ret
```

Esercizio 2

Scrivere una funzione che crea un file archivio in cui inserisce un numero arbitrario di file in modo simile a quanto avviene per un file tar.

La funzione deve avere il seguente prototipo:

- `archive` è il pathname del file archivio di output (es. `archive.dat`)
- `files` è un array di stringhe che rappresentano i pathname dei file di input da archiviare in archive
- n è il numero di file da archiviare

Ogni file archiviato ha una `header` formata da:

- 256 byte che contengono il pathname del file (come stringa C, quindi con terminatore zero alla fine del pathname - non è necessario ovviamente che il pathname effettivo usi tutti i 256 byte disponibili e i byte extra saranno padding)
- 8 byte che rappresentano la dimensione del file.

Alla `header` seguono i byte del file stesso.

Se un file con il pathname `archive` esiste già, il suo contenuto deve essere inizialmente troncato a dimensione zero dalla funzione `archiver`. Il file archivio creato deve avere privilegi di lettura e scrittura per l'utente proprietario, sola lettura per il gruppo proprietario, e nessun permesso per tutti gli altri.

Suggerimenti: Per calcolare la dimensione size di un file `fd` si può usare `size = lseek(fd, 0, SEEK_END)`. La system call si posiziona alla fine del file e restituisce l'offset corrente che coincide con la dimensione in byte del file stesso. Per scopi di debugging, è possibile esplorare il contenuto binario del file di output archive.dat creato dal programma di prova con il comando `xxd archive.dat | less`

```c
#include "e2.h"
#include <stdio.h>
#include <stdlib.h>
#include <unistd.h>
#include <string.h>
#include <sys/types.h>
#include <sys/stat.h>
#include <fcntl.h>
#define FILENAME_LEN 256
#define BUF_SIZE 4096

void check_error(long res, char* msg) {
    if (res != -1) return;
    perror(msg);
    exit(EXIT_FAILURE);
}

void copy_file(int fd_dest, int fd_src) {
    ssize_t res;
    char buf[BUF_SIZE];
    for (;;) {
        res = read(fd_src, buf, BUF_SIZE);
        check_error(res, "read");
        if (res == 0) break;
        res = write(fd_dest, buf, res);
        check_error(res, "write");
    }
}
```

```c
void archiver(const char* archive, const char** files, int n) {
    char filename[FILENAME_LEN];
    int fd_archive, fd, i;
    long res;

    fd_archive = open(archive, O_WRONLY | O_CREAT | O_TRUNC, 0644);
    check_error(fd_archive, "open");

    for (i=0; i<n; ++i) {
        fd = open(files[i], O_RDONLY);
        check_error(fd, "open");

        long size = lseek(fd, 0, SEEK_END);
        check_error(size, "lseek");

        res = lseek(fd, 0, SEEK_SET);
        check_error(res, "lseek");

        strcpy(filename, files[i]);

        res = write(fd_archive, filename, FILENAME_LEN);
        check_error(res, "write");

        res = write(fd_archive, &size, sizeof(size));
        check_error(res, "write");

        copy_file(fd_archive, fd);

        res = close(fd);
        check_error(res, "close");
    }

    res = close(fd_archive);
    check_error(res, "close");
}
```