- 7° Modulo
  - Funzioni Parte 2^ - b
    - Esempi
    - Liste
  - _____
  - _____
  - _____

http://www.elet.polimi.it/upload/martucci/index.html

lista

```
#include <malloc.h>
#include <stdlib.h>
#include <stdio.h>
#define  NULL 0

typedef struct element {DATIIMPORTANTI  vs;
                        struct element   *next;
                        }ELEMENT ;

ELEMENT *create_list_element();
void add_element( ELEMENT *e);

/*  ------------------------------  */
/* Il puntatore di inizio della lista "head" è esterno alle funzioni
   ed è visibile da tutte quante, come pure il tipo ELEMENT  */
/*  ------------------------------  */

ELEMENT *head =NULL;

/*  ------------------------------  */
/* per 10 volte aggiunge un elemento in fondo ad una lista */
/*  ------------------------------  */
void main( )
{
  int j;

  for (j=0; j < 10; ++j)
    add_element ( create_list_element() );
};

/*  ------------------------------  */
/* ogni volta crea un elemento (controlli!!!) e restituisce il puntatore  */
/*  ------------------------------  */
ELEMENT *create_list_element()
{
  ELEMENT *p;

  p = (ELEMENT *) malloc( sizeof ( ELEMENT ) );
  if (p == NULL)
  {
    printf( "create_list_element: malloc failed.\n");
    exit( 1 );
  }
  p->next = NULL;
  return p;
};


/*  ------------------------------  */
/* ogni volta riceve un puntatore ad un singolo elemento
   collegato a NULL e lo inserisce in fondo */
/*  ------------------------------  */

void add_element( ELEMENT *e)
{
  ELEMENT *p;
  /* If the first element (the head) has not been
   * created, create it now.
   */
  if (head == NULL)
  {
    head = e;
    return;
  }
  /* Otherwise, find the last element in the list */
  for (p = head; p->next != NULL; p = p->next)
    ; /* null statement */
  p->next = e;
 };
```

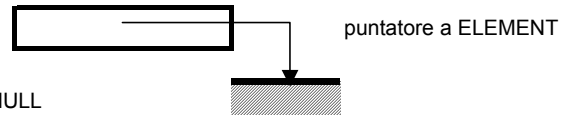
```
#include <malloc.h>
#include <stdlib.h>
#include <stdio.h>
#define  NULL 0
```

ELEMENT è un tipo

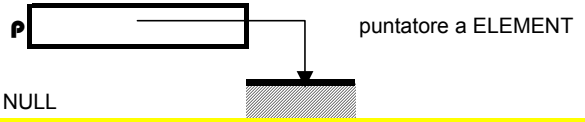
```
typedef struct element {DATIIMPORTANTI  vs;
                        struct element   *next;
                        }ELEMENT ;
```


```
ELEMENT *create_list_element();
void add_element( ELEMENT *e);
```

```
ELEMENT *head =NULL;
```

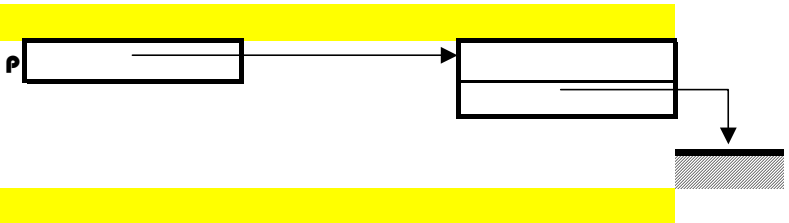



```
ELEMENT *create_list_element()
{
  ELEMENT *p;
```




```
  p = (ELEMENT *) malloc( sizeof ( ELEMENT ) );
```


```
  if (p == NULL)
  {
    printf( "create_list_element: malloc failed.\n");
    exit( 1 );
  }
  p->next = NULL;
```

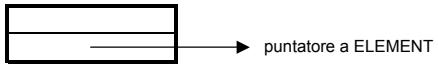


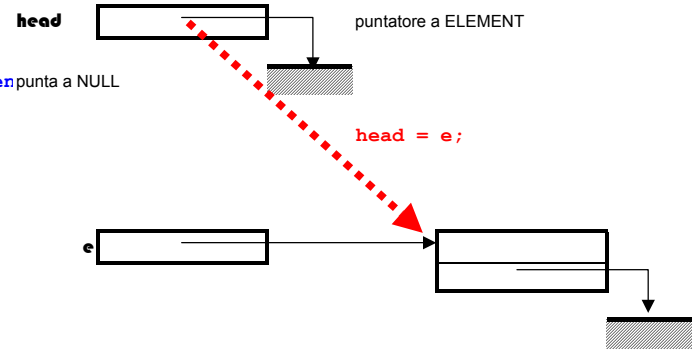

```
  return p;
};
```

p è il risultato

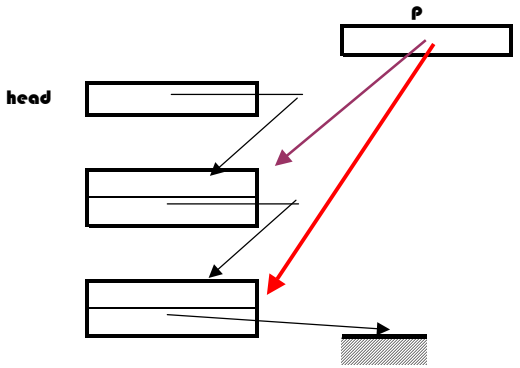
```
#include <malloc.h>
#include <stdlib.h>
#include <stdio.h>
#define  NULL 0

typedef struct element {DATIIMPORTANTI  vs;
                        struct element   *next;
                        }ELEMENT ;

ELEMENT *create_list_element();
void add_element( ELEMENT *e);

ELEMENT *head =NULL;


void add_element( ELEMENT *e)
{
        /* inserisce un elemento in fondo alla lista */

  ELEMENT *p;
  /* If the first element (the head) has not been
   * created, create it now.
   */
  if (head == NULL)
  {
    head = e;
    return;
  }
```

ELEMENT è un tipo



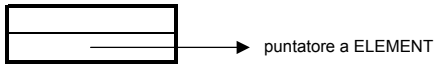


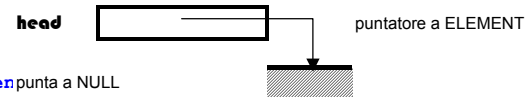

```
  /* Otherwise, find the last element in the list */
  for (p = head; p->next != NULL; p = p->next)
    ; /* null statement */
};


  p->next = e;
};
```



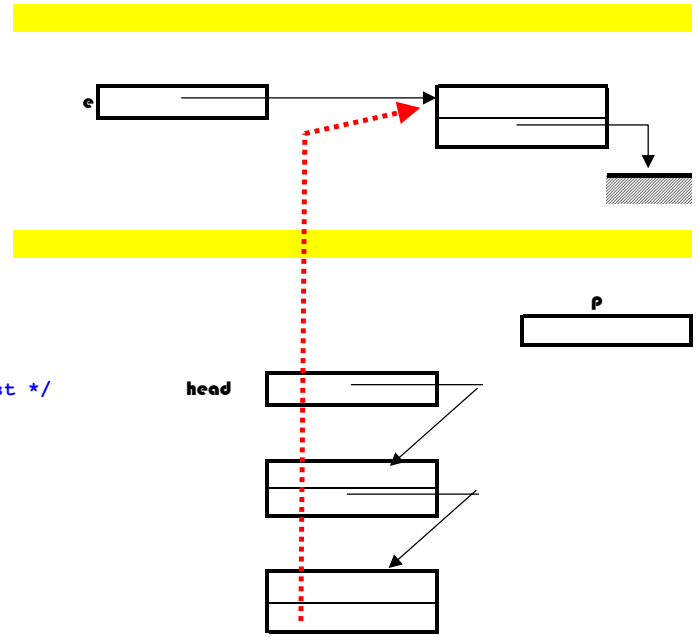
```
#include <malloc.h>
#include <stdlib.h>
#include <stdio.h>
#define  NULL 0

typedef struct element {DATIIMPORTANTI  vs;
                        struct element   *next;
                        }ELEMENT ;

ELEMENT *create_list_element();
void add_element( ELEMENT *e);

ELEMENT *head =NULL;


void add_element( ELEMENT *e)
{
        /* inserisce un elemento in fondo alla lista */

  ELEMENT *p;
  /* If the first element (the head) has not been
   * created, create it now.
   */
  if (head == NULL)
  {
    head = e;
    return;
  }

  /* Otherwise, find the last element in the list */
  for (p = head; p->next != NULL; p = p->next)
    ; /* null statement */
};


  p->next = e;
};
```

ELEMENT è un tipo





punta a NULL



lista

```
#include <malloc.h>
#include <stdlib.h>
#include <stdio.h>
#define  NULL 0

typedef struct element {DATIIMPORTANTI  vs;
                        struct element   *next;
                        }ELEMENT ;

ELEMENT *create_list_element();
void add_element( ELEMENT *e);

ELEMENT *head =NULL;


void main( )
{
  int j;

  for (j=0; j < 10; ++j)
    add_element ( create_list_element() );
};

/*  ------------------------------  */

ELEMENT *create_list_element()
{
  ELEMENT *p;

  p = (ELEMENT *) malloc( sizeof ( ELEMENT ) );
  if (p == NULL)
  {
    printf( "create_list_element: malloc failed.\n");
    exit( 1 );
  }
  p->next = NULL;
  return p;
};
```

Esempio – parametri formali/attuali: Scambio di valore (SWAP) tra due variabili

```
/* Swap the values of
two int variables */
void swap( int *x, int *y)
{
  int temp;
  temp = *x;
  *x = *y;
  *y = temp;
}
```

```
int main( void )
{
  int a = 2, b = 3;
  swap ( &a, &b );
  printf( "a = %d\t b = %d\n",
                    a, b );
}
```

Il valore di Ritorno

- Deve essere presente il tipo altrimenti è **`int`** (NON **`void`**!!!)
- Il valore restituito deve essere UN SOLO VALORE (con la istruzione **`return`**)
- Il valore non può essere array
- E' possbile passare più di un valore mediante puntatore a un tipo aggregato
- Una **`struct`** o **`union`** possono essere passati, ma il passaggio avviene per ricopiatura e può essere inefficiente

Le regole di conversione della return sono come quelle dell'assegnamento

```
char *f(){ /*restituisce puntatore a carattere*/
  char **cpp, *cp1, *cp2, ca[10];    int *ip1;
  cp1 = cp2;              /* OK, types match */
  return cp2;             /* OK, types match */
  cp1 = *cpp;             /* OK, types match */
  return *cpp;            /* OK, types match */
/* An array name without a subscript gets converted
   * to a pointer to the first element.   */
  cp1 = ca;               /* OK, types match */
  return ca;              /* OK, types match */
  cp1 = *cp2;             /* Error, mismatched types */
                          /* (pointer to char vs. char) */
  return *cp2;            /* Error, mismatched types */
                          /* (pointer to char vs. char) */
  cp1 = ip1;              /* Error, mismatched pointer types */
  return ip1;             /* Error, mismatched pointer types */
  return;                 /* Produces undefined behavior    */
                          /* should return (char *) */
}
```

Sample C Code

decl vs no decl

```
main()
{
   double x, y;

   x = 4.0;
   y = sqrt(x);
   printf("f\n", y);
}
```

No declaration

prints wrong answer of 16640.000000, because the double returned is presumed to be an int!

```
main()
{
   double x, y;
   double sqrt(double);

   x = 4.0;
   y = sqrt(x);
   printf("%f\n", y);
}
```

with declaration

prints correct answer of 2.000000

Pointers to Functions

Declaring pointers to functions

cannot omit parenthesis

int (*p)(); p is ptr to function that returns an int

int *p(); p is function that returns ptr to int

assign address of function as with array

```
int fname(void);
int (*fnptr)(void);
fnptr = fname;
```

fname *è un indirizzo!!!!!*

Precedenze

- Usare questa tabella per controllare i prossimi esempi!!

Level	Operator
16L	-> . [] ()
15R	sizeof ++ -- ~ ! + - (cast)
	*** indiretto &indirizzo**
13L	* / %
12L	+ -
11L	<< >>
10L	< <= > >=
9L	== !=
8L	& and bitwise
7L	^ xor bitwise
6L	| or bitwise
5L	&& AND logico
4L	|| OR logico
2R	= *= /= %= += -= <<= >>= &= |= ^=
1L	, virgola

```
Declaration            Description
====================   ==================================================

int *p;                /* p is a pointer to an integer quantity */

int *p[10];            /* p is a 10 element array of pointers to int */

int (*p)[10];          /* p is a pointer to a 10 element array of int */

int *p(void);          /* p is a function that returns a pointer to int */

int p(char *a);        /* p is a function that accepts an argument of
                          a pointer to char and returns an int */

int *p(char *a);       /* p is a function that accepts an argument of a
                          pointer to char and returns a pointer to int */

int (*p)(char *a);     /* p is a pointer to a function that accepts a pointer
                          to char as an argument and returns an int */

int (*p(char *a))[10]; /* p is a function that accepts an argument which is
                          a pointer to char and returns a pointer to a
                          10-element array of int */

int p(char (*a)[]);    /* p is a function that accepts an argument which is
                          a pointer to an array of char and returns an int */

int p(char *a[]);      /* p is a function that accepts an argument which is
                          an array of pointers to char and returns an int */
```

```
Declaration              Description
====================     ==================================================

int *p(char a[]);        /* p is a function that accepts an argument which is
                            an array of char and returns a pointer to an int */

int *p(char (*a)[]);     /* p is a function that accepts an argument which is
                            a pointer to an array of char and returns a pointer
                            to an int */

int *p(char *a[]);       /* p is a function that accepts an argument which is
                            an array of pointer to char and returns a pointer
                            to an int */

int (*p)(char (*a)[]);   /* p is a pointer to a function that accepts an
                            argument which is a pointer to an array of char and
                            returns an int */

int *(*p)(char (*a)[]); /* p is a pointer to a function that accepts an
                            argument which is a pointer to an array of char and
                            returns a pointer to an int */

int *(*p)(char *a[]);    /* p is a pointer to a function that accepts an
                            argument which is an array of pointers to char and
                            returns a pointer to an int */

int (*p[10])(void);      /* p is a 10 element array of pointers to functions;
                            each function returns an int */

int (*p[10])(char a);    /* p is a 10 element array of pointers to functions;
                            each function accepts an argument which is a char
                            and returns an int */

int *(*p[10])(char a);   /* p is a 10 element array of pointers to functions;
                            each function accepts an argument which is a char
                            and returns a pointer to an int */
```

```
int *(*p[10])(char *a); /* p is a 10 element array of pointers to functions;
                           each function accepts an argument which is a pointer
                           to a char and returns a pointer to an int */
```

AMBIENTI, VISIBILITÀ DELLE VARIABILI

AMBIENTE GLOBALE DEL PROGRAMMA

- insieme di identificatori (tipi, costanti, variabili) definiti nella parte dichiarativa globale
- **regole di visibilità**: visibili a tutte le funzioni del programma

AMBIENTE LOCALE DI UNA FUNZIONE

- insieme di identificatori (tipi, costanti, variabili) definiti nella parte dichiarativa locale **e** degli identificatori definiti nella testata (parametri formali)
- **regole di visibilità**: visibili alla funzione e ai blocchi in essa contenuti

AMBIENTE DI BLOCCO

- insieme di identificatori (tipi, costanti, variabili) definiti nella parte dichiarativa locale di blocco
- **regole di visibilità**: visibili al blocco e ai blocchi contenuti

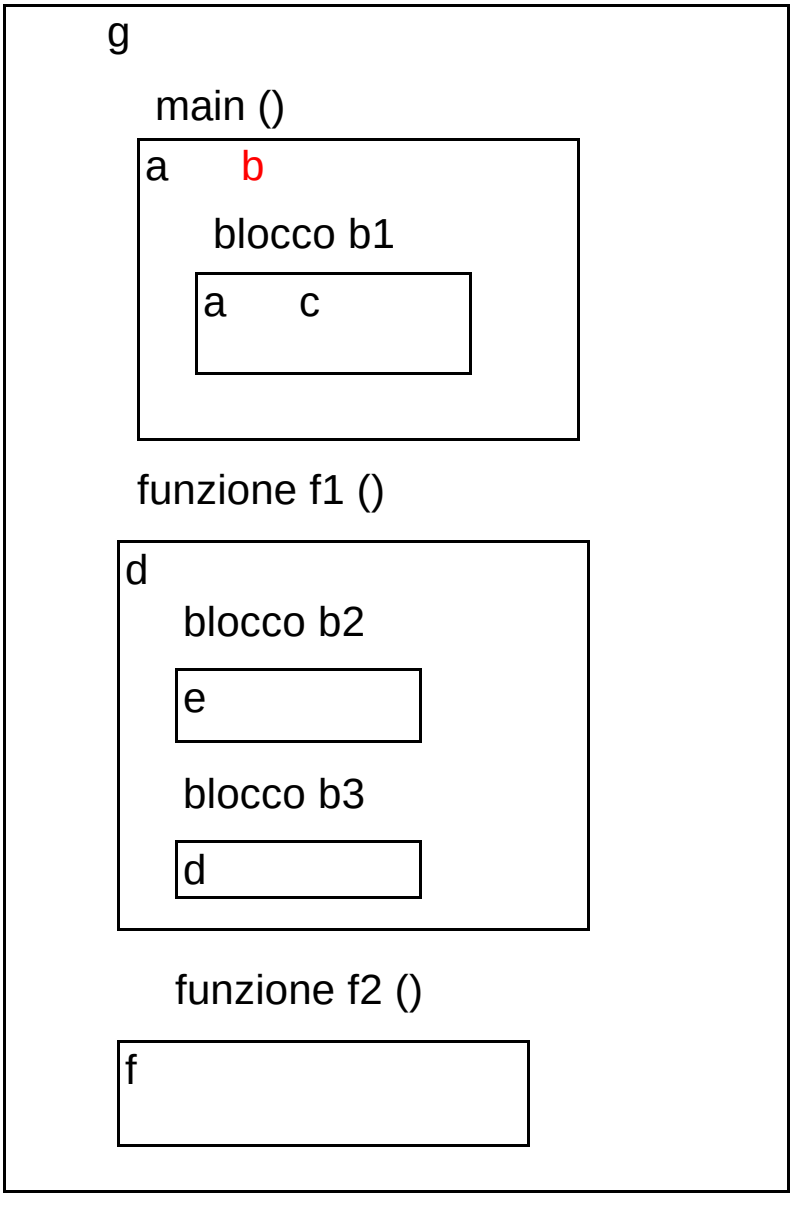
IDENTIFICATORE DI FUNZIONE:

- visibile a tutte le funzioni e a se stessa

OMONIMIA DI IDENTIFICATORI IN AMBIENTI DIVERSI

- è visibile quello dell’ambiente più «vicino»

ESEMPIO DI AMBIENTI E REGOLE DI VISIBILITÀ



main ()
- g
- a, b
- f1, f2

blocco b1
- g
- b (di main)
- a, c
- f1, f2

funzione f1()
- g
- d
- f1, f2

blocco b2
- g
- d (di f1)
- e
- f1, f2

blocco b3
- g
- d (di b3)
- f1, f2

funzione f2()
- g
- f
- f1, f2

Sample C Code

nested blocks

This is legal, but can be confusing!

exists from here...

'exist' only within here

to here

```
main()
{
    int x = 1, y = 2;
    printf("x = %d, y = %d\n", x, y);
    {
        int x = 3, y = 4;
        printf("x = %d, y = %d\n", x, y);
    }
    printf("x = %d, y = %d\n", x, y);
}
```

This 'x' and 'y' ...

are not the same variables ...

as this 'x' and 'y' !

but these are!

Output is:

```
x = 1, y = 2
x = 3, y = 4
x = 1, y = 2
```

AMBIENTE DI ESECUZIONE DI UNA FUNZIONE

Sequenza di chiamate

main → f1 →f2
main ← f1 ←

- L'**ambiente di esecuzione** di una funzione viene creato al momento della chiamata e «rilasciato» solo quando la funzione termina.
- In una **sequenza di chiamate** l'ultimo chiamato è il primo a terminare.

RICORSIONE:

la soluzione ad un problema si dice ricorsiva (l'algoritmo è ricorsivo) se fa uso di se stessa.

La soluzione ad un problema (l'algoritmo) può ammettere o non ammettere formulazione ricorsiva.

IN PROGRAMMAZIONE:

dato un sottoprogramma P, la chiamata di P quando P è in esecuzione (P chiama se stesso) è ricorsione.

Per distinguere le varie chiamate si parla di **attivazione** del sottoprogramma

main →P'→ P''→ P'''
main ←P'← P''←

La chiamata di P può anche essere indiretta.

main →P'→ Q→ P''
main ←P'← Q←

AMBIENTE DI ESECUZIONE E RICORSIONE

Un linguaggio di programmazione (e quindi l'ambiente di programmazione) può supportare o meno la ricorsione. **Il C ammette la ricorsione.**

Perché la ricorsione sia possibile:

- l'ambiente di esecuzione deve essere associato all'attivazione di P (nasce un ambiente per ogni attivazione e tale ambiente viene rilasciato al termine dell'attivazione considerata)

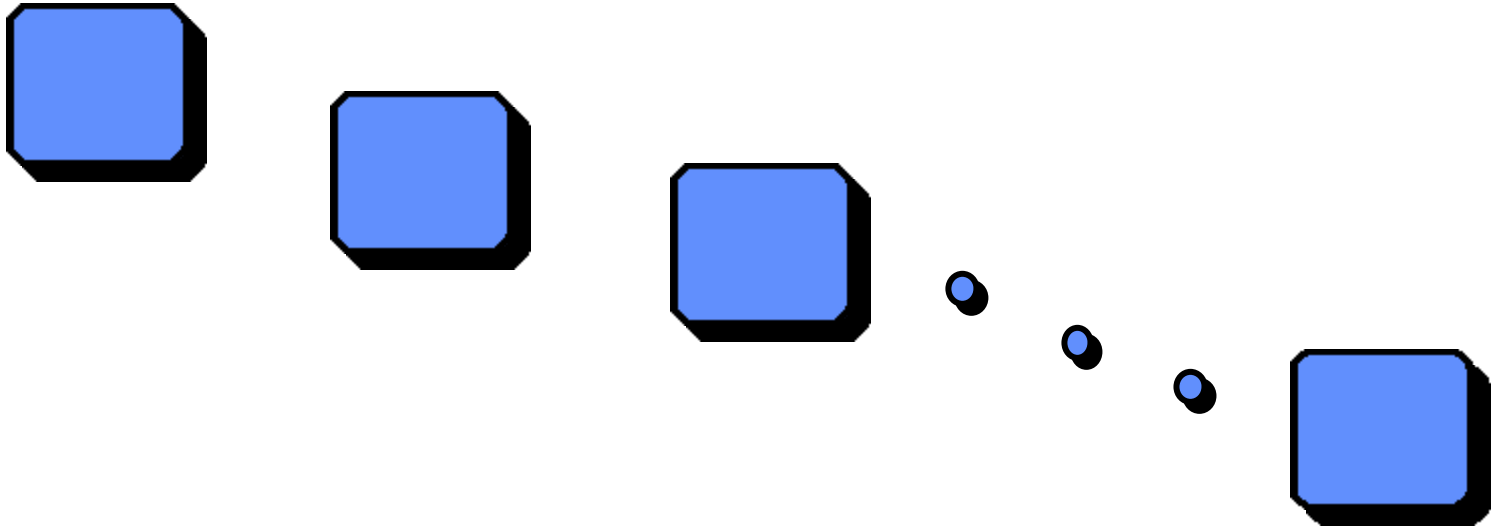
E' necessaria una opportuna gestione della memoria di lavoro allocata a contenere l'ambiente di esecuzione di un sottoprogramma: gestione a pila (stack) degli ambienti di esecuzione.

RECORD DI ATTIVAZIONE: è costituito da un'area di memoria adeguata a contenere

- l'ambiente locale della funzione (dichiarazioni locali e parametri formali)
- l'indirizzo di ritorno al chiamante
- informazioni per gestire la memoria a stack

Recursion

- A Recursive function is one that calls itself
- Sometimes recursion is most natural solution
- Higher overhead with multiple function calls
- Need to be able to recognize recursion



Recursion

Problem: compute a factorial (of positive integers)

Analysis: What is a factorial?

n! = n * (n-1) * (n-2) * ... 1

with 1! defined to be 1

and 0! defined to be 1

base case: when n = 1

recursion step: n * (factorial of n -1)

Sample C Code

recursive

```
#include <stdio.h>
long factorial(long);

main()
{
   int j;

   for ( j = 0; j <= 10; j++)
      printf("%2d! = %ld\n", j, factorial(j));
}

long factorial(long number)
{
   if( number <= 1)
      return 1;
   else
      return( number * factorial(number - 1));
}
```

Base case

recursion step

Recursione

- Funzione recursiva: chiama sé stessa.

  - Appropriate for iterative processes where each step mimics another, e.g., tree searches
  - E.g., factorial calculation: $n! = n(n-1)!$

```
// why a long?
unsigned long factorial (unsigned int number) {
    if (number > 1)
        return (number * factorial (number - 1));
    return 1ul;    // for 0 or 1
}
```

  - Matches the problem (i.e., the mathematics) nicely

Recursive vs. Iterative Functions

```
// iterative version
unsigned long factorial_iter (unsigned int number) {
    for (unsigned long product = 1ul; number > 1; number --)
        product *= number;
    return product;
}
```

- Recursive vs. iterative
  - smaller
  - less complex
  - but, slower—due to additional function invocations

Unless otherwise required, do what is most natural.

RECORD DI ATTIVAZIONE: FUNZIONAMENTO

- ad ogni attivazione viene allocato un record di attivazione
- al termine dell'attivazione il record viene rilasciato (l'area di memoria è riutilizzabile)
- la dimensione del record di attivazione di una funzione è nota e fissa e viene determinata in fase di compilazione
- non è noto il numero di chiamate (di attivazioni) della funzione: tale numero dipende dall'esecuzione del programma
- i record vengono allocati in memoria «a pila» (stack): «uno sopra l'altro» e il primo record dello stack è relativo all'ultima funzione attivata e non ancora terminata. Lo stack «cresce» dal basso verso l'alto
- il primo record di attivazione è allocato per la funzione main().

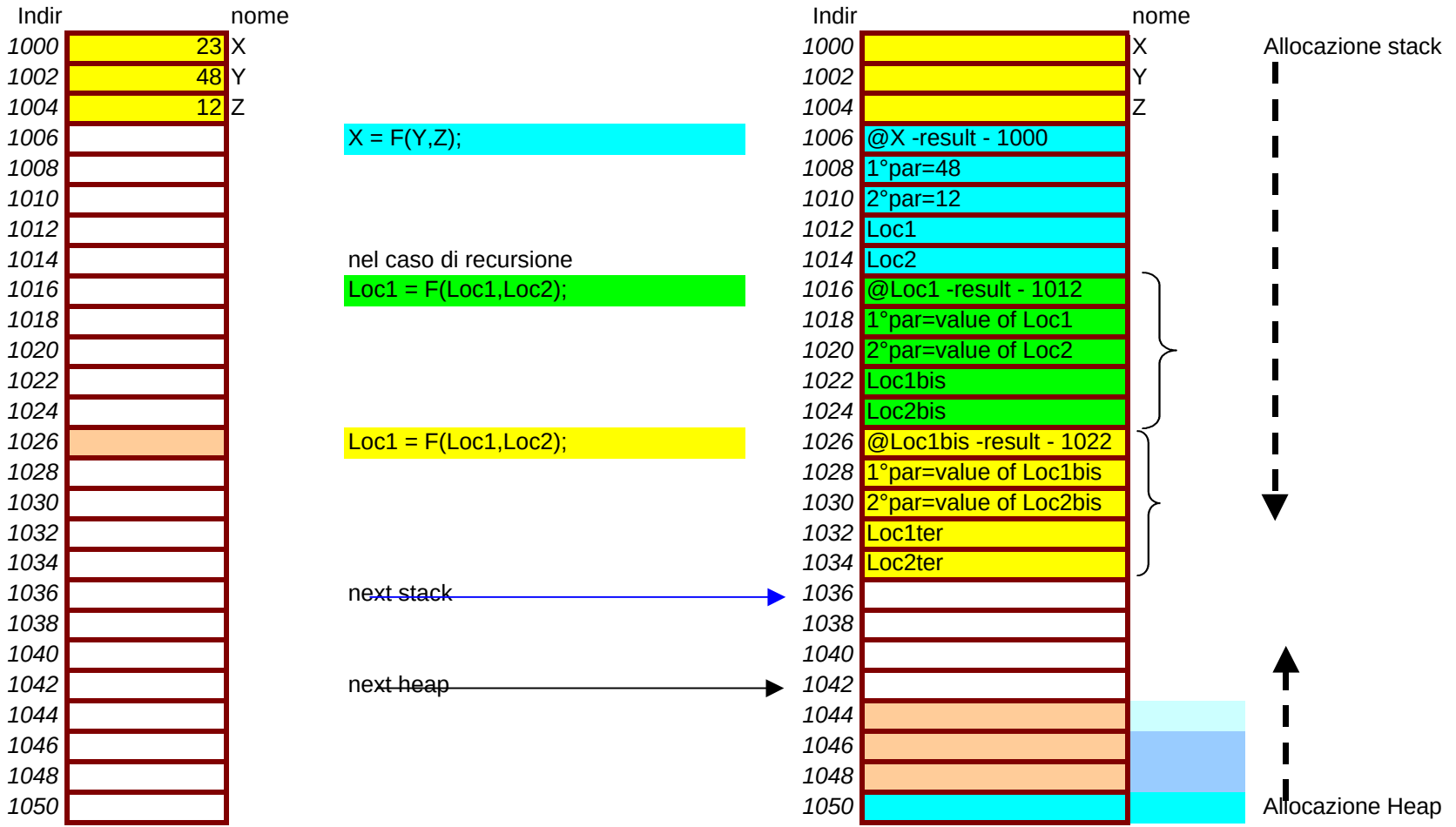
INFORMAZIONI PER GESTIRE LA MEMORIA A STACK

stack pointer: è l'indirizzo della cima della pila
in ogni record di attivazione è memorizzato il valore dello stack pointer relativo a quell'attivazione specifica

RAM E STACK OVERFLOW

Una parte della RAM è dedicata a contenere l'area di stack, che ha globalmente delle dimensioni prefissate. Si parla di **stack overflow** quando questa capacità viene superata («troppi» annidamenti di chiamate).

```
int X,Y,Z;
int F (int, int);
```



ESEMPIO: CONVERSIONE BINARIA RICORSIVA

```
#include <stdio.h>
#define TRUE 1
#define FALSE 0
void main ()
{
 int valore;
 char proseguire,continua,tappo;

 void Converti_bin(int);

 proseguire=TRUE;

 while (proseguire)
  {
  printf("inserire il valore intero positivo  da convertire \n");
   scanf("%d",&valore);

   Converti_bin(valore);

   printf("\nVuoi convertire un altro valore? (S/N) \n");
   scanf("%c",&continua);
   scanf("%c",&tappo);
   if (continua!='S')
      proseguire=FALSE;
  }
} /* fine main */
```

```
void Converti_bin (int num)
{ int resto;

  resto=num%2;

  if (num >=2)
     Converti_bin (num/2);

  printf("%d", resto);
}
```