Programmazione 2: Nozioni Generali

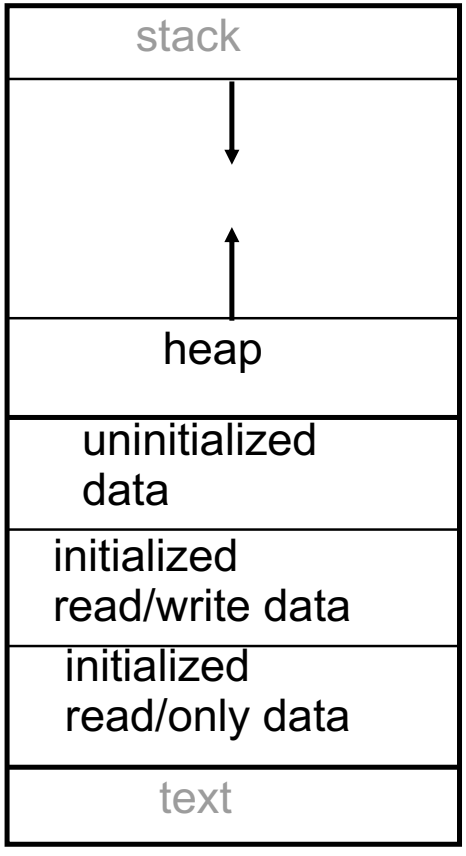
Marco Patrignani[1]

mailto: marco.patrignani@unitn.it
website: https://squera.github.io/

1 UNIVERSITÀ DI TRENTO

Memoria

Il modello di memoria



memoria allocata dalle funzioni (Variabili automatiche)

memoria allocata dinamicamente dal programmatore

Variabili globali e statiche

<- questo é supportato solo da alcuni hardware

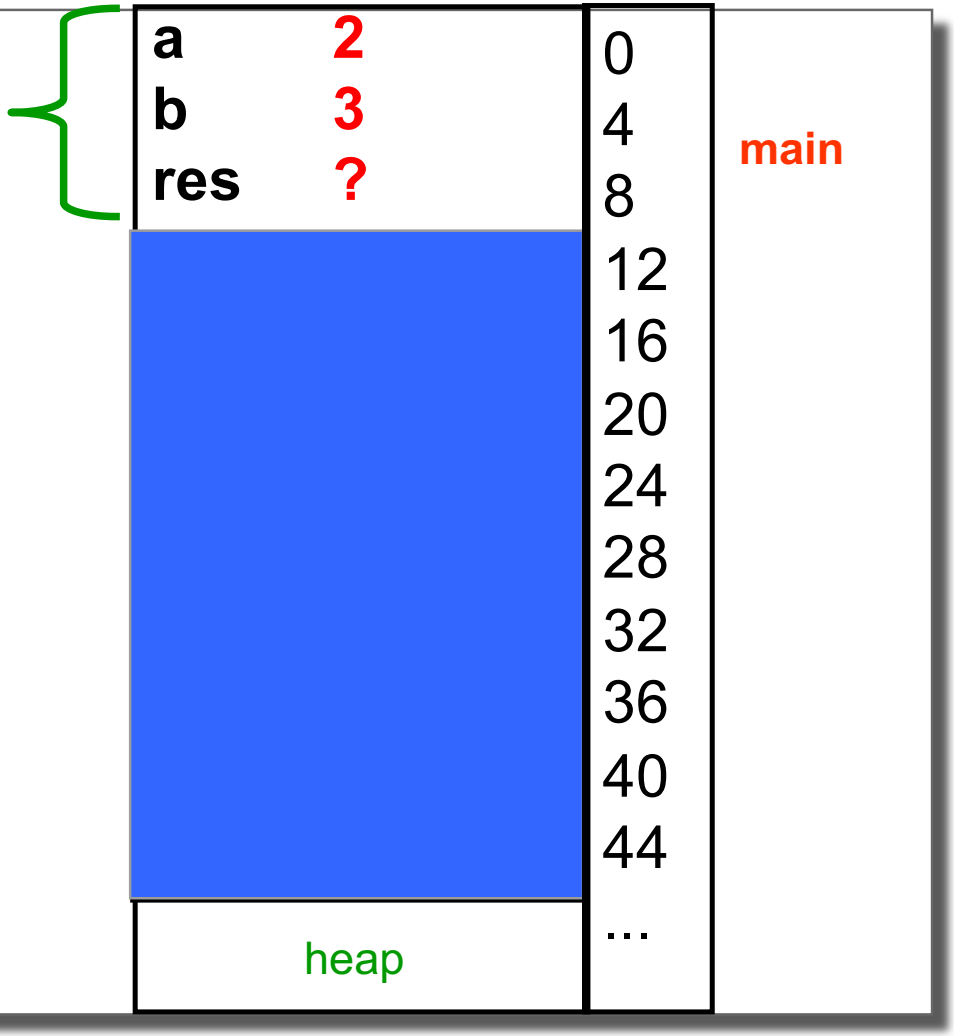
Codice eseguibile

Funzioni, Stack e Ricorsione

Modularizzazione: Funzioni

Esempio

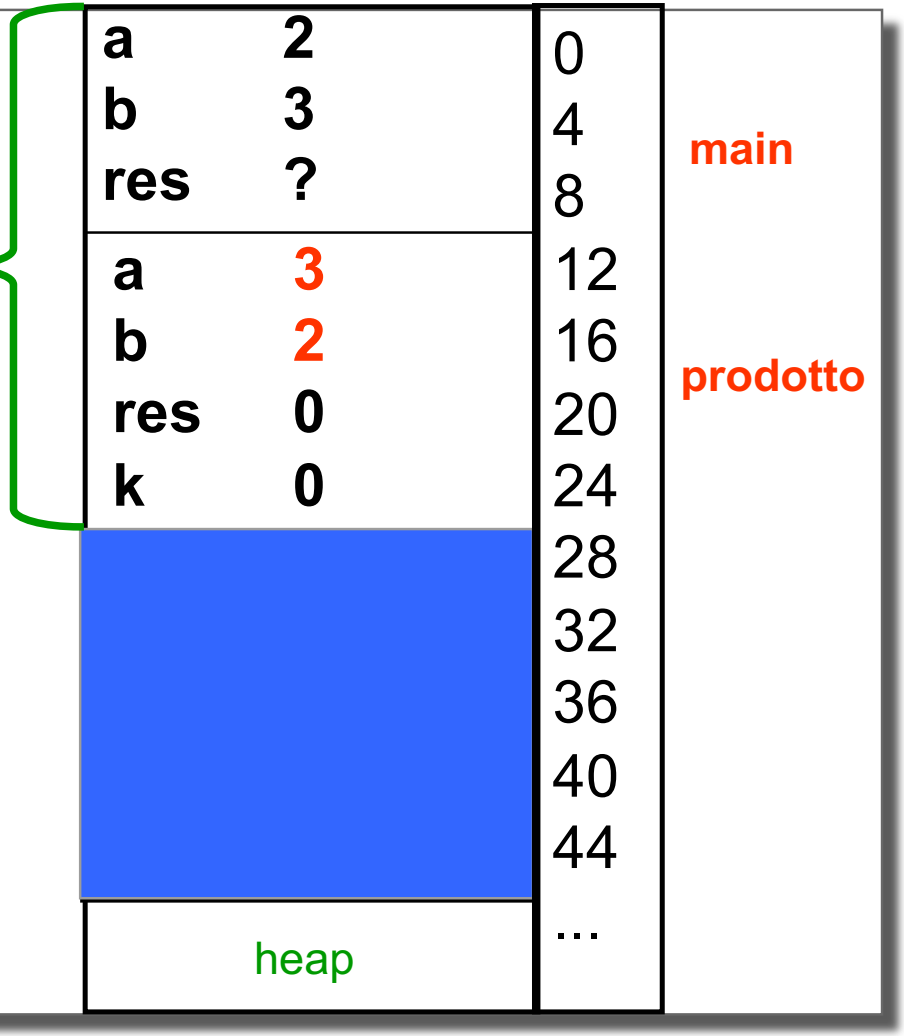
```
int somma(int a, int b) {
    int res;
    res=a+b;
    return res;
}
int prodotto(int b, int a) {
    int res=0;
    for (int k=0; k<b; k++)
       res=somma(res,a);
    return res;
}
main() {
    int a,b,res;
    cout << "dammi due numeri \n";
    cin >> a >> b;
    res=prodotto(a,b);
    cout << a << " * " << b << " =
" << res << "\n";
}
```



Modularizzazione: Funzioni

Esempio

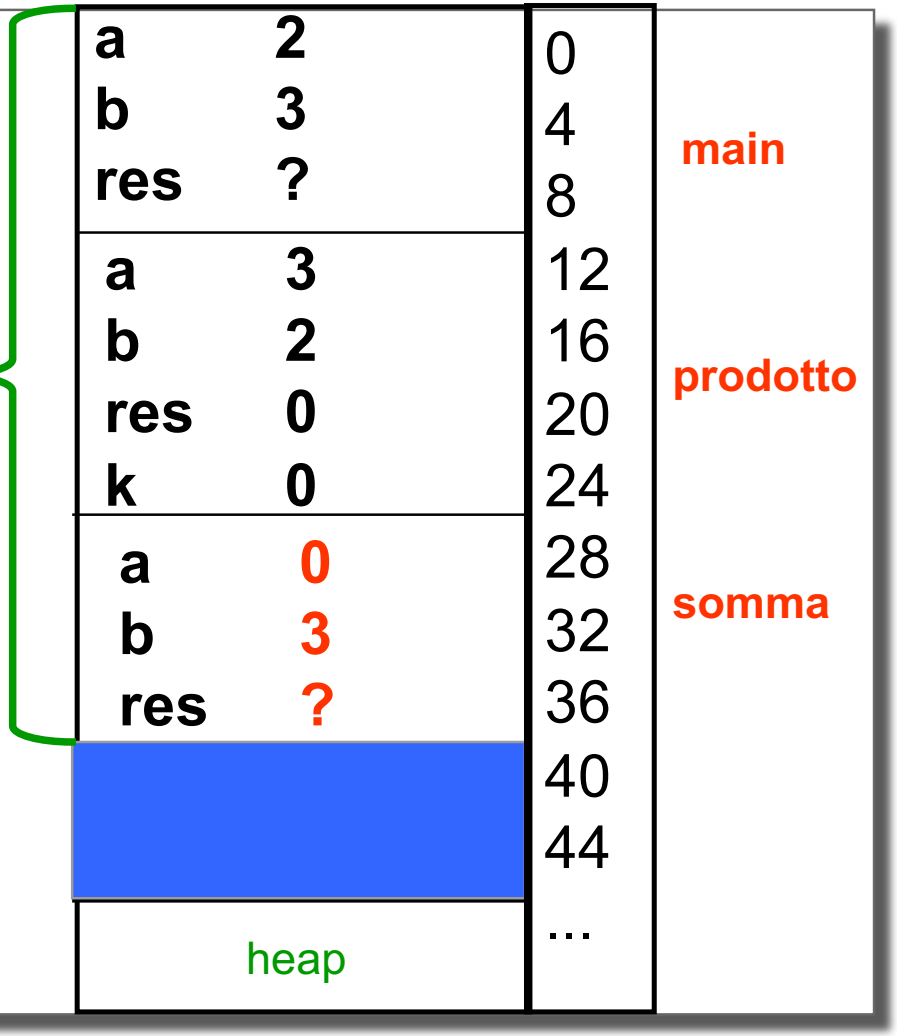
```
int somma(int a, int b) {
    int res;
    res=a+b;
    return res;
}
int prodotto(int b, int a) {
    int res=0;
    for (int k=0; k<b; k++)
        res=somma(res,a);
    return res;
}
main() {
    int a,b,res;
    cout << "dammi due numeri \n";
    cin >> a >> b;
    res=prodotto(a,b);
    cout << a << " * " << b << " =
" << res << "\n";
}
```



Modularizzazione: Funzioni

Esempio

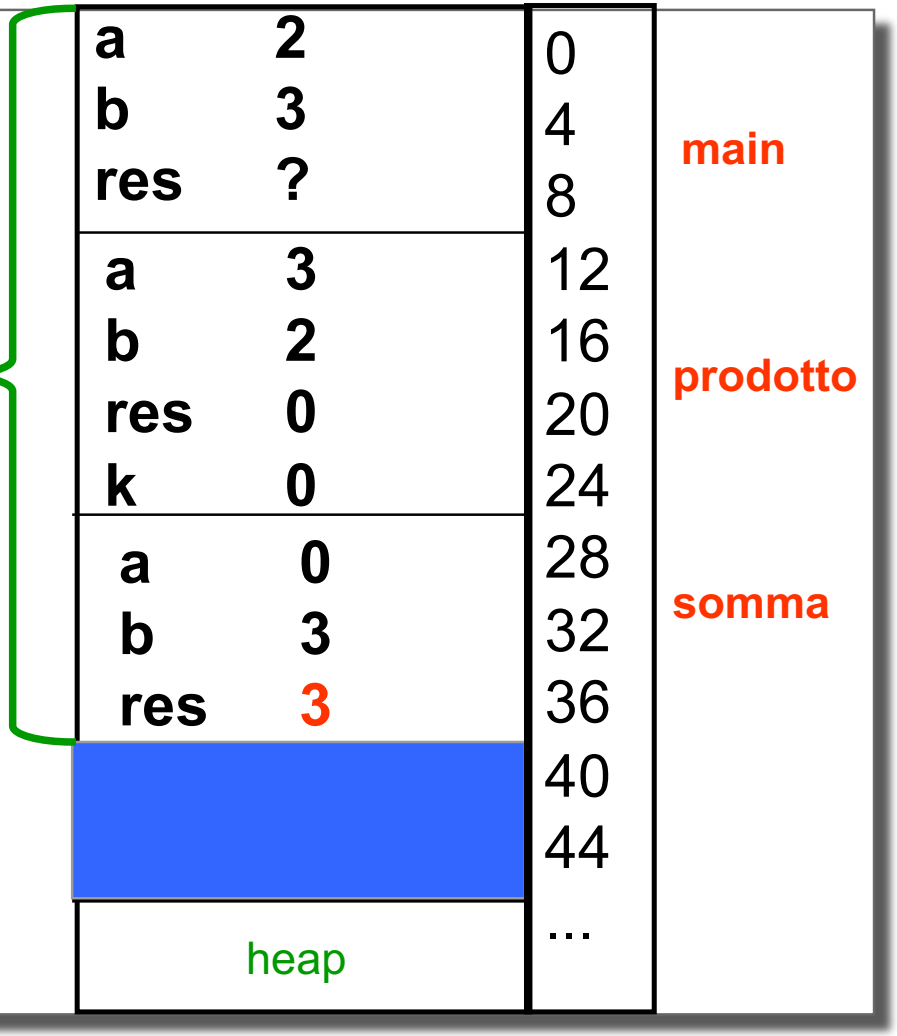
```
int somma(int a, int b) {
    int res;
    res=a+b;
    return res;
}
int prodotto(int b, int a) {
    int res=0;
    for (int k=0; k<b; k++)
        res=somma(res,a);
    return res;
}
main() {
    int a,b,res;
    cout << "dammi due numeri \n";
    cin >> a >> b;
    res=prodotto(a,b);
    cout << a << " * " << b << " =
" << res << "\n";
}
```



Modularizzazione: Funzioni

Esempio

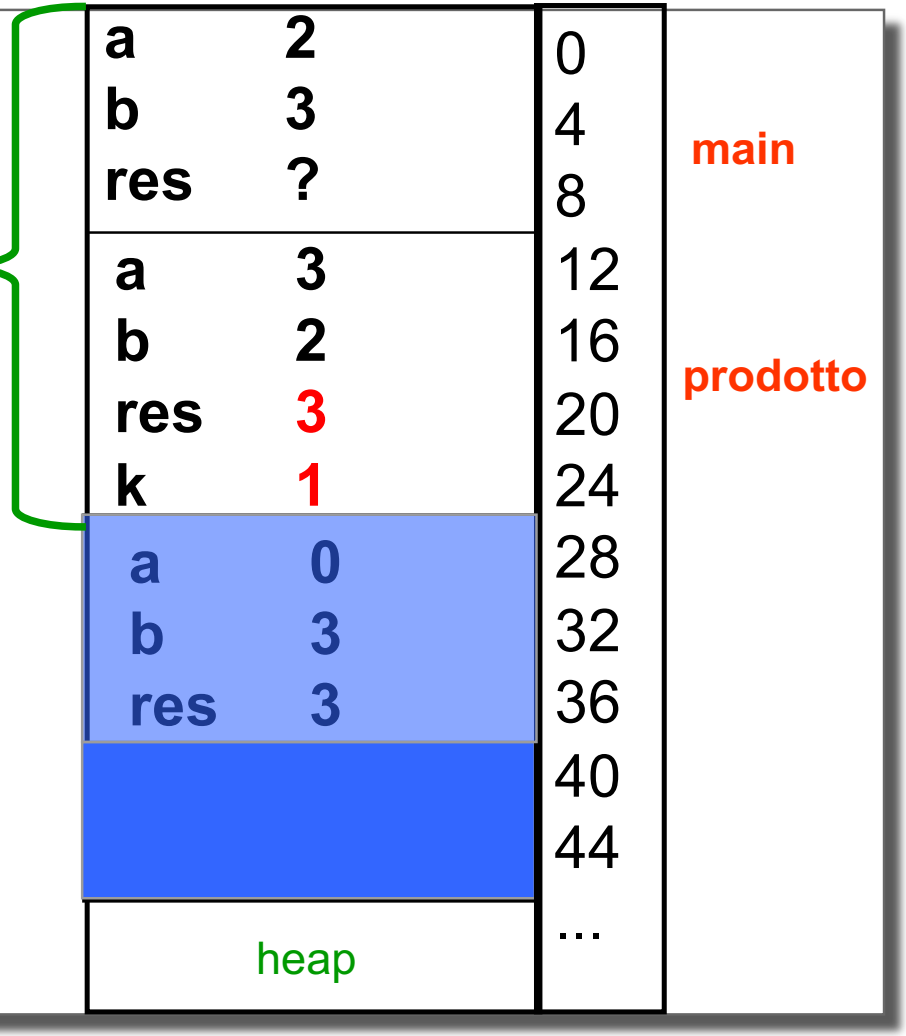
```
int somma(int a, int b) {
    int res;
    res=a+b;
    return res;
}
int prodotto(int b, int a) {
    int res=0;
    for (int k=0; k<b; k++)
       res=somma(res,a);
    return res;
}
main() {
    int a,b,res;
    cout << "dammi due numeri \n";
    cin >> a >> b;
    res=prodotto(a,b);
    cout << a << " * " << b << " =
" << res << "\n";
}
```



Modularizzazione: Funzioni

Esempio

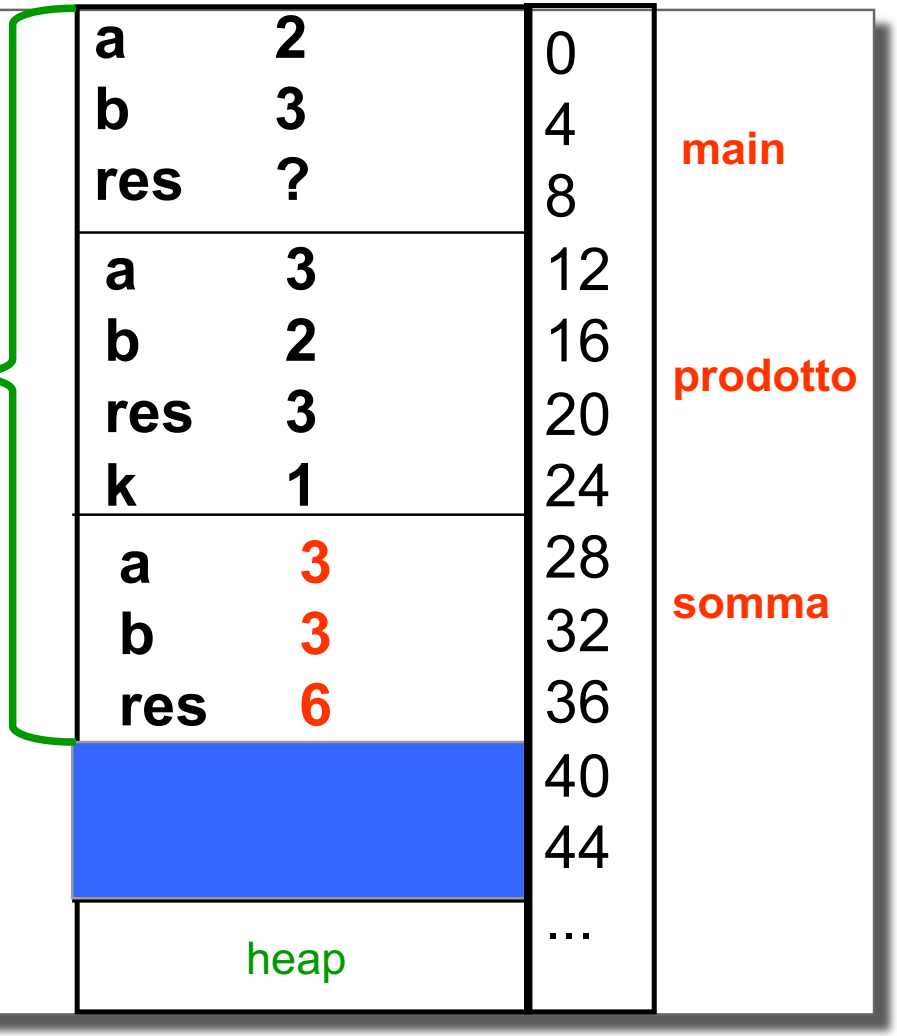
```
int somma(int a, int b) {
    int res;
    res=a+b;
    return res;
}
int prodotto(int b, int a) {
    int res=0;
    for (int k=0; k<b; k++)
        res=somma(res,a);
    return res;
}
main() {
    int a,b,res;
    cout << "dammi due numeri \n";
    cin >> a >> b;
    res=prodotto(a,b);
    cout << a << " * " << b << " =
" << res << "\n";
}
```



Modularizzazione: Funzioni

Esempio

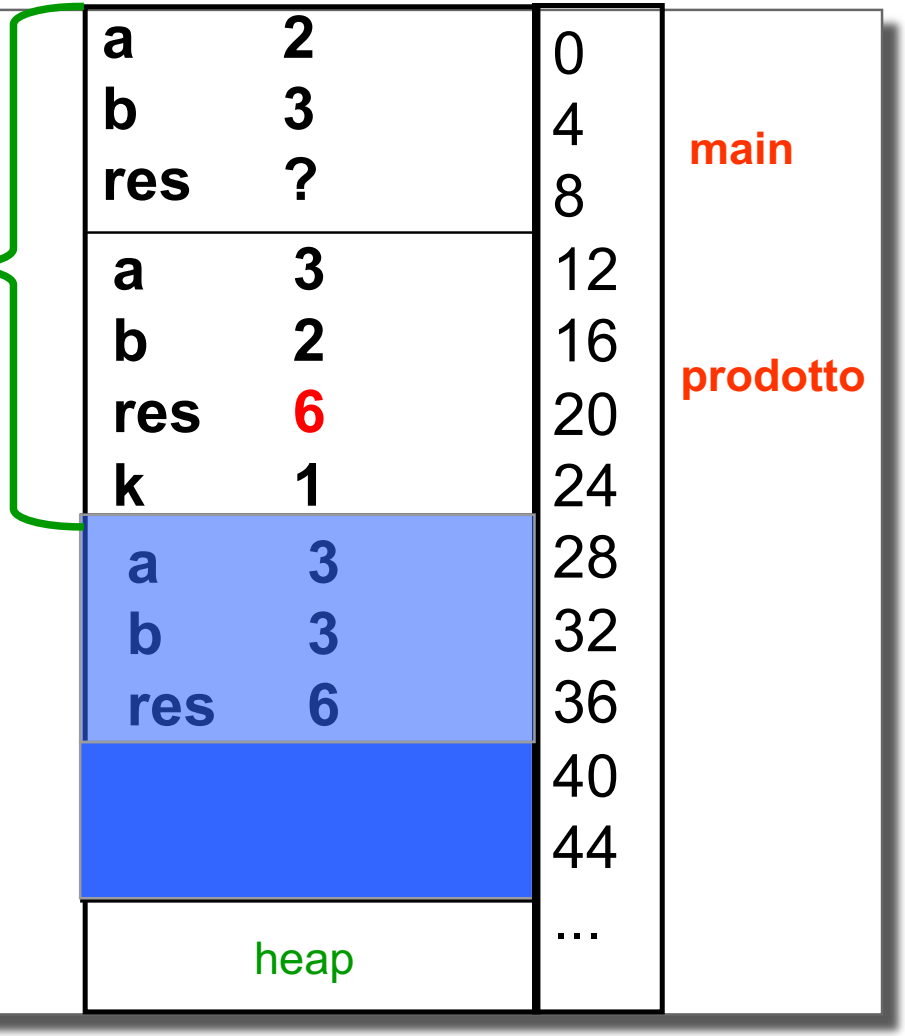
```
int somma(int a, int b) {
    int res;
    res=a+b;
    return res;
}
int prodotto(int b, int a) {
    int res=0;
    for (int k=0; k<b; k++)
        res=somma(res,a);
    return res;
}
main() {
    int a,b,res;
    cout << "dammi due numeri \n";
    cin >> a >> b;
    res=prodotto(a,b);
    cout << a << " * " << b << " =
" << res << "\n";
}
```



Modularizzazione: Funzioni

Esempio

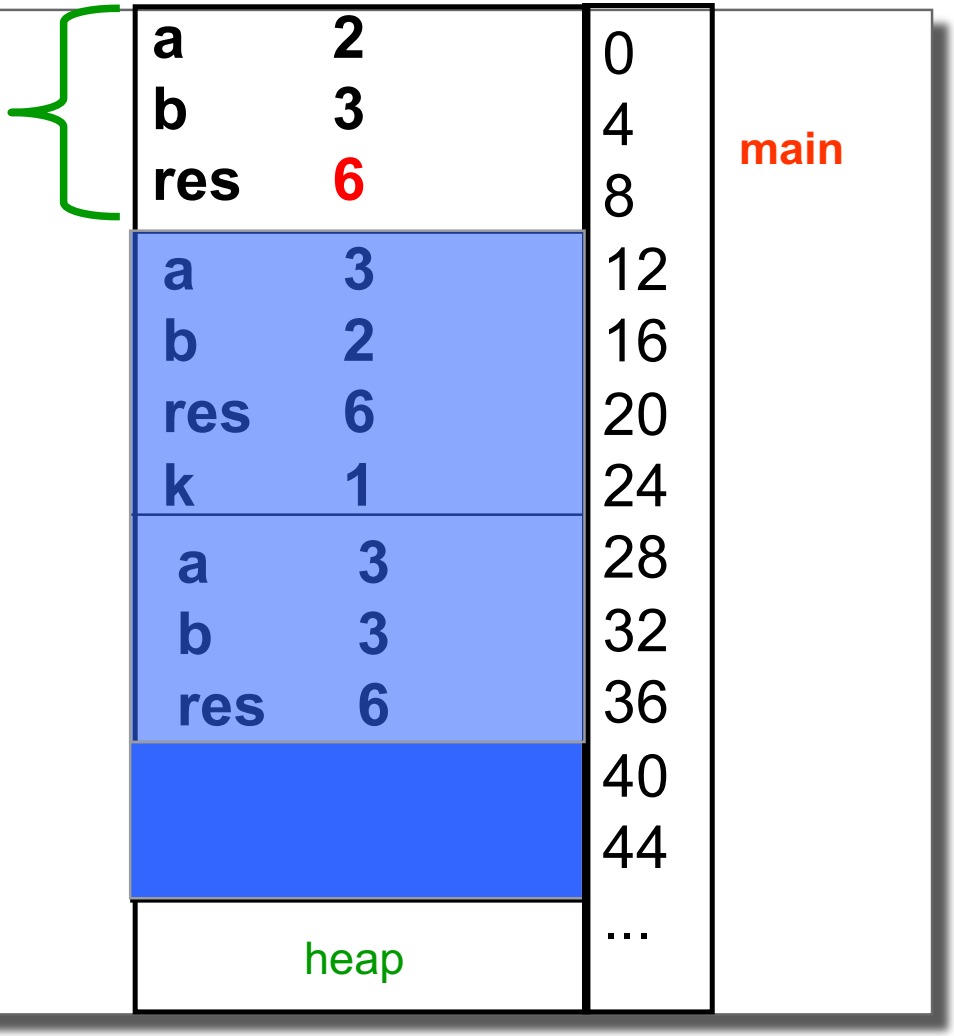
```
int somma(int a, int b) {
    int res;
    res=a+b;
    return res;
}
int prodotto(int b, int a) {
    int res=0;
    for (int k=0; k<b; k++)
        res=somma(res,a);
    return res;
}
main() {
    int a,b,res;
    cout << "dammi due numeri \n";
    cin >> a >> b;
    res=prodotto(a,b);
    cout << a << " * " << b << " =
" << res << "\n";
}
```



Modularizzazione: Funzioni

Esempio

```
int somma(int a, int b) {
    int res;
    res=a+b;
    return res;
}
int prodotto(int b, int a) {
    int res=0;
    for (int k=0; k<b; k++)
       res=somma(res,a);
    return res;
}
main() {
    int a,b,res;
    cout << "dammi due numeri \n";
    cin >> a >> b;
    res=prodotto(a,b);
    cout << a << " * " << b << " =
" << res << "\n";
}
```



Funzioni ricorsive

Una funzione può richiamare se stessa.

```
int fact(int n) {
    if (n==0) return 1;
    else return n*fact(n-1);
}
main(void) {
   int n;
   cout<<"dammi un numero\n;
   cin >> n;
   cout << "Il suo fattoriale vale "<<fact(n)<<"\n";
}
```

Cosa avviene nello stack?

Variabili

Scope delle variabili: le globali

Nel seguente esempio a è una variabile globale.
Il suo valore è visibile a tutte le funzioni.

Le variabili globali vanno EVITATE a causa dei side-effects.

```
int a=5;
void f() {
   a=a+1;
   cout << "a in f: " << a << " - ";
   return;
}
main() {
   cout << "a in main:" << a << " - ";
   f();
   cout << "a in main: " << a << endl);
}
```

Output:

```
a in main: 5  - a in f: 6 - a in main: 6
```

Scope delle variabili: le automatiche

Nel seguente esempio a e' una variabile automatica per la funzione f.
Il suo valore è locale ad f.

```
int a=5;
void f() {
   int a=2, b=4;
   printf("(a,b) in f: (%d,%d) -",a,b);
   return;
}
main() {
   int b=6;
   printf("(a,b) in main: (%d,%d) -",a,b);
   f();
   printf("(a,b) in main: (%d,%d)\n",a,b);
}
```

Output:

```
(a,b) in main: (5,6)  - (a,b) in f: (2,4) - (a,b) in main: (5,6)
```

ATTENZIONE! Le variabili automatiche SCHERMANO le variabili globali.

Quanto vale s?

```
void modifica(int s) {
     s++;;
}
main(void) {
    int s=1;
    modifica(s);
    cout << "s=" << s << "\n";
}
```

<– A) "locale"

"globale" (B–>

```
int s;
void modifica() {
     s++;
}
main(void) {
    s=1;
    modifica();
    cout << "s=" << s << "\n";
}
```

Variabili globali

Le variabili globali sono "cattive"

(almeno quanto il GOTO)!

perchè violano il principio della località della informazione (Principio di "**Information hiding**")

E' impossibile gestire correttamente progetti "grossi" nei quali si faccia uso di variabili globali.

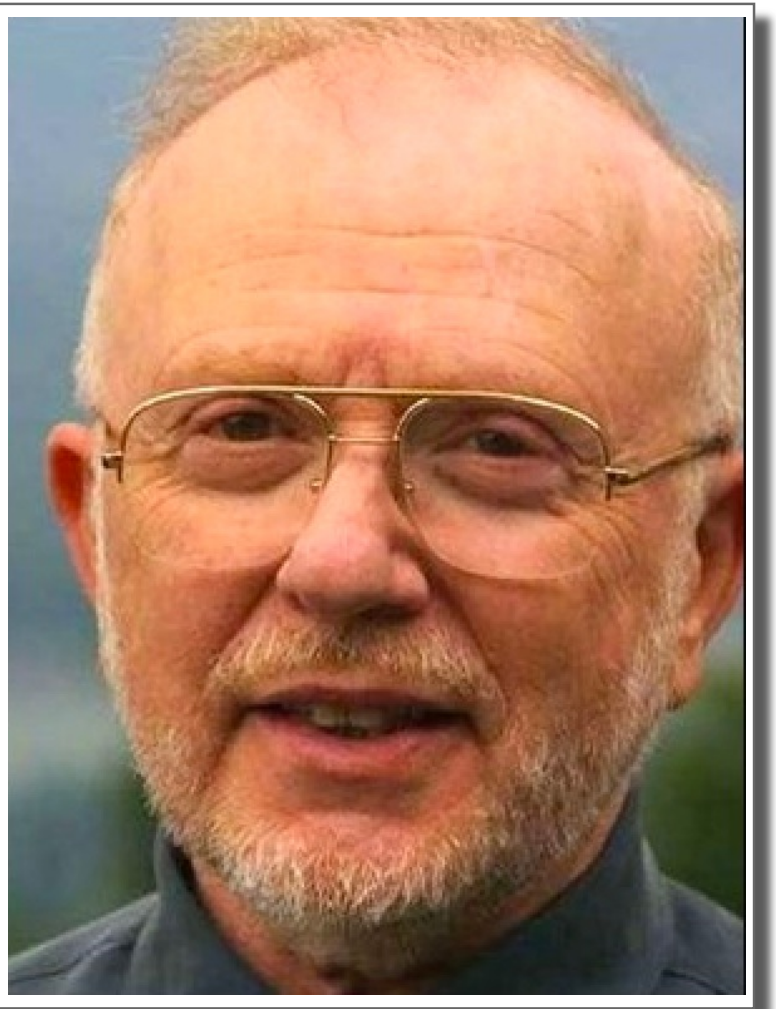
Principio del **NEED TO KNOW**:

Ciascuno deve avere **TUTTE** e **SOLO** le informazioni che servono a svogere il compito affidato

Principi di Parnas

Il committente di una funzione deve dare all'implementatore tutte le informazioni necessarie a realizzare la funzione,
e NULLA PIÙ

L'implementatore di una funzione deve dare all'utente tutte le informazioni necessarie ad usare la funzione,
e NULLA PIÙ



David Parnas

Calling Convention

Funzioni: problema #1

```
void incrementa(int x) {
     x=x+1;
}
main(void) {
    int a=1;
    incrementa(a);
    cout << "a=" a << "\n";
}
```

Come faccio a scrivere una funzione che modifichi le variabili del chiamante?

Quanto vale a quando viene stampata?

I parametri sono passati per valore (copia)!

Funzioni: problema #2

Come faccio a farmi restituire
più di un valore da una funzione?

Puntatori

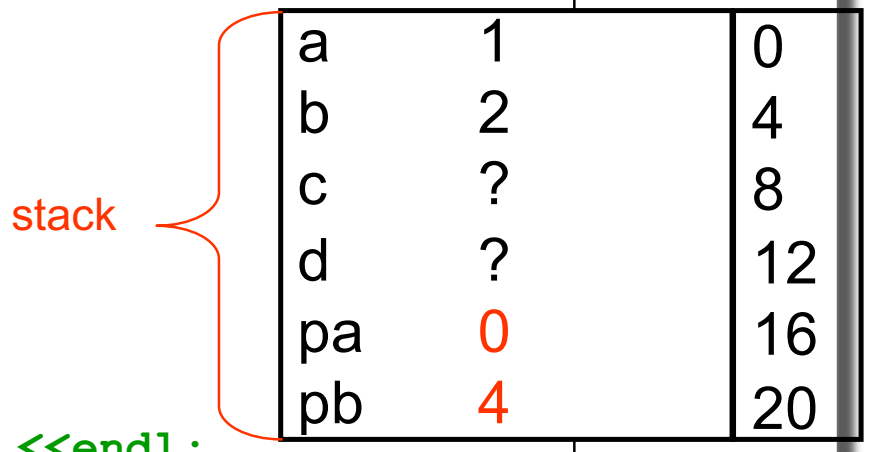
Operatore indirizzo: &

&a fornisce l'indirizzo della variabile a

Operat. di dereferenziazione: *

*p interpreta la variabile p come un puntatore (indirizzo) e fornisce il valore contenuto nella cella di memoria puntata

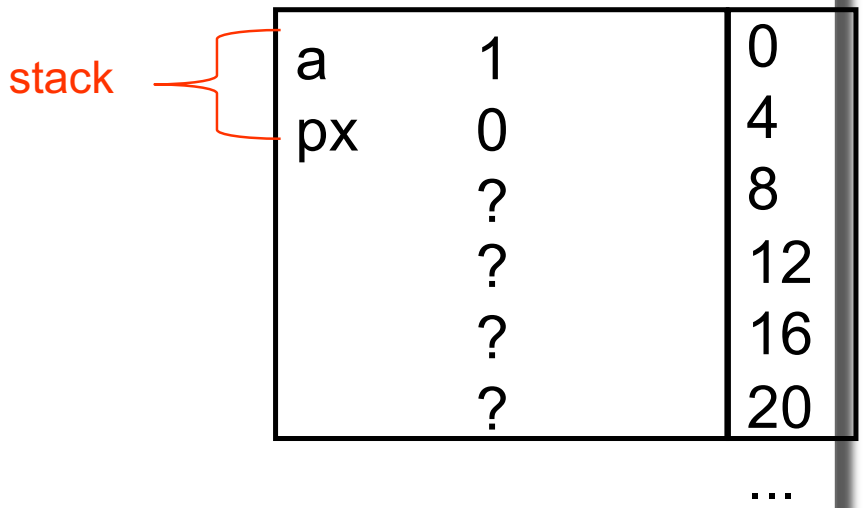
```
main() {
      int a,b,c,d;
      int * pa, * pb;
      pa=&a; pb=&b;
      a=1; b=2;
      c=a+b;
      d=*pa + *pb;
      cout << a<<"  "<<b<<"  "<< c <<endl;
      cout << a <<"  "<< *pb <<"  "<< d <<endl;
}
```



Funzioni e puntatori

TRUCCO: per passare un parametro per indirizzo,
passiamo per valore un puntatore ad esso!

```
void incrementa(int *px) {
     *px=*px+1;
}
main(void) {
    int a=1;
    incrementa(&a);
    cout<<a<<endl;
}
```



OUTPUT: 2

Array, Puntatori e Heap

Array

Gli array sono collezioni di elementi omogenei

```
int valori[10];
char v[200], coll[4000];
```

Un array di *k* elementi di tipo *T* in è un blocco di memoria contiguo di grandezza

```
(k*sizeof(T))
```

Array - 2

Ogni singolo elemento di un array può essere utilizzato esattamente come una variabile con la notazione:

```
valori[indice]
```

dove indice stabilisce quale posizione considerare all'interno dell'array

Limitazioni

Gli indici spaziano sempre tra *0* e *k-1*

Il numero di elementi è fisso (deciso a livello di compilazione - *compile time*): non può variare durante l'esecuzione (a *run time*)

Non c'è nessun controllo sugli indici durante l'esecuzione

Catastrofe (potenziale)

```
....
int a[10];
a[256]=40;
a[-12]=50;
....
```

NOTA: le stringhe sono array di char

Operatori *new* e *delete*

`new type` alloca **`sizeof(type)`** bytes in memoria (heap) e restituisce un puntatore alla base della memoria allocata. (esiste una funzione simile usata in C e chiamata **`malloc`**)

`delete(* p)` dealloca la memoria puntata dal puntatore p. (Funziona solo con memoria dinamica allocata tramite new. Esiste un'analoga funzione in C chiamata **`free`**).

Il mancato uso della **`delete`** provoca un insidioso tipo di errore: il **memory leak**.

Allocazione della memoria

Allocazione statica di memoria (at compile time)

```
main() {
    int a;
    cout<<a<<endl; //NO!
    a=3;
    cout<<a<<endl;
}
```

```
OUTPUT:    1
           3
```

Allocazione dinamica di memoria (at run time)

```
main() {
    int *pa;
    pa=new int;
    cout<<*pa<<endl; //NO!
    *pa=3;
    cout<<*pa<<endl;
    delete(pa);
    cout<<*pa<<endl; //NO!
}
```

```
OUTPUT:    4322472
           3
           8126664
```

Vettori rivistati

Dichiarare un vettore è in un certo senso come dichiarare un puntatore.

```
v[0]    è equivalente a    *v
```

Attenzione però alla differenza!

```
int v[100];    è "equivalente" a:
                    int *v; v=new int[100];
```

ATTENZIONE!
la prima versione alloca spazio STATICAMENTE (Stack)
la seconda versione alloca spazio DINAMICAMENTE (Heap)