1

Programmazione II

Marco Ronchetti

2

Obiettivi

Il corso introduce le tecniche e costrutti della programmazione ad oggetti come una evoluzione necessaria per affrontare il problema della crescente complessità degli artefatti software.

Verranno utilizzati i linguaggi C++ e Java.

Il corso è prevalentemente teorico: vi saranno alcune esercitazioni di introduzione a tool per l'uso di Java.

3

Impegno

1 credito = 25 ore di studio

6 crediti = 150 ore. In aula: 7x7=49 ore

⇒PER OGNI ORA DI LEZIONE IN AULA OCCORRE STUDIARE (Studio, ripasso, esercizi) DUE ORE FUORI AULA

4

Supporto

Materiale on-line

- *copia delle slides*
- *diario delle lezioni*
- *registrazione audio-video delle lezioni (on line – su CD)*
- *forum di discussione*

Info su : http://latemar.science.unitn.it

5

Supporto

Registrazione audio-video delle lezioni

Software "e-presence"
del KMDI – University of Toronto

6

Supporto

Registrazione audio-video delle lezioni:
requisiti

On line (diretta o differita):
- *modem 56 Kb:* — *Audio+Slide only*
- *connessione ISDN 128 Kb* — *Video,Audio,Slide*
- *connessione ADSL* — *Video,Audio,Slide*

Off line:
- *CD player*

SW: Real Player (free)

7

Logistica

Domani 17 febbraio:
sciopero contro il disegno di legge Moratti di riforma dell'Università

Le ore di lezione saranno comunque recuperate successivamente.

8

Logistica

Orario:
Lezione (4 o 7 ore per settimana)
Esercitazione a gruppi
(0 o 3 ore per settimana)

Settimana	1	2	3	4	5	6	7
Gruppo	A	B	C	A/B/C	A	B	C

Composizione dei gruppi:

A	B	C
A-D	E-P	Q-Z
		Apprendistato

9

Programmazione industriale

Programming "in the large"

- *Suddivisione del lavoro tra persone/gruppi (divide et impera)*
- *Mantenibilità (che succede se voglio cambiare qualcosa tra un mese/un anno/...)*
- *Robustezza (che succede se sostituisco una persona?)*

10

Programmazione industriale

Le risposte:

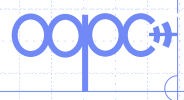
Ingegneria del software
(corso del prossimo anno)

Buone tecniche di programmazione
(es. commenti up to date)

Supporto dal linguaggio:
Object Oriented Programming
(in C++)
(in Java)

11

Richiami di C++ di base



12

Cominciamo con una domanda…

Se dichiaro

```
int k;
```

qual'e' il massimo valore che posso immagazzinare nella variabile k?

13

Operatore *sizeof*

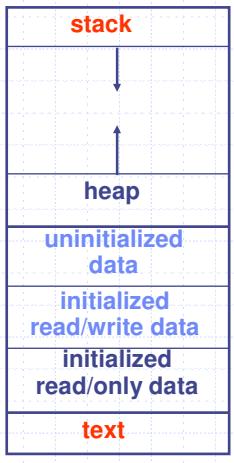
- **sizeof(tipo)** restituisce il numero di bytes necessari a immagazzinare il tipo specificato
- **sizeof(var)** controlla il tipo di var e restituisce il numero di bytes necessari al tipo a cui var appartiene.

```
#include <iostream.h>
main() {
    int a=0;
    cout<< "dimensione di un intero : "<<sizeof(int)<<"\n";
    cout<< "dimensione della variabile a : "<<sizeof(a)<<"\n";
}
```

```
Output:
dimensione di un intero : 4
dimensione della variabile a : 4
```

14

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

15

Modularizzazione: Funzioni

Funzioni come "procedure"

```
void stampa(void) {
    const char EOL="\n";
    cout << "sono arrivato qui"<<EOL;
}

main() {
    int a=0;
    a++;
    stampa();
    a++;
    stampa();
    a++;
    stampa();
}
```

Nota: il main è una funzione!

16

Modularizzazione: Funzioni

Funzioni come "procedure parametrizzate"

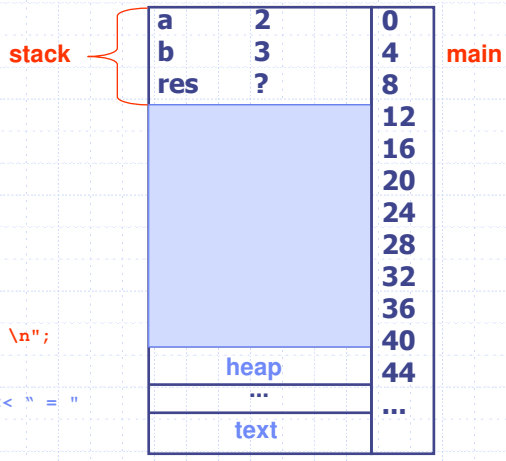
```
tipo funzione(tipo argom1,...,tipo argomN)
 {
        corpo della funzione
        return var;
 }
```

17

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";
}
```

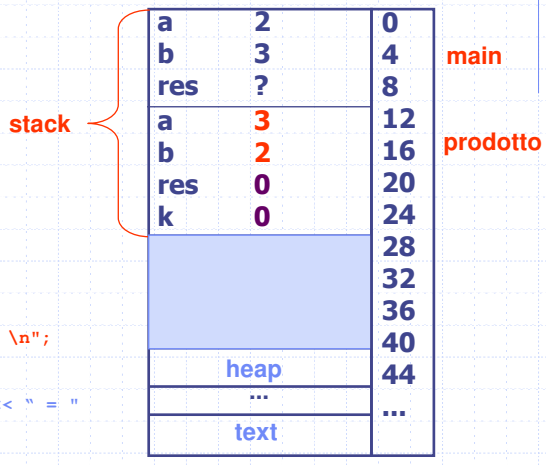


18

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";
}
```



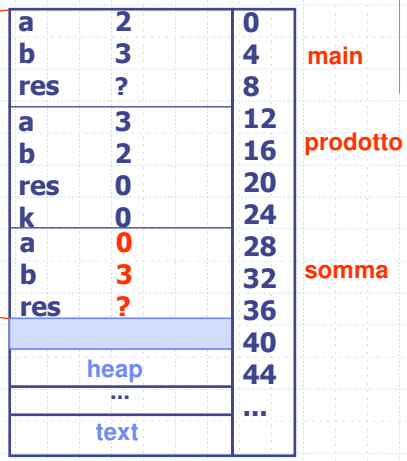
19

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";
}
```

stack



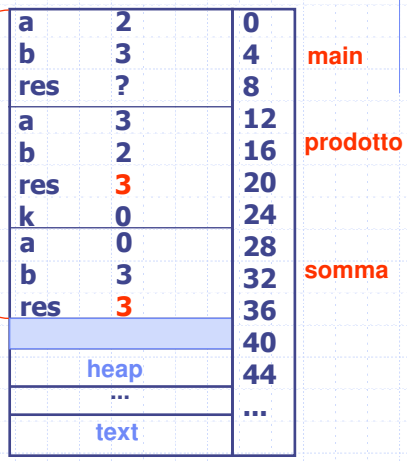
20

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";
}
```

stack

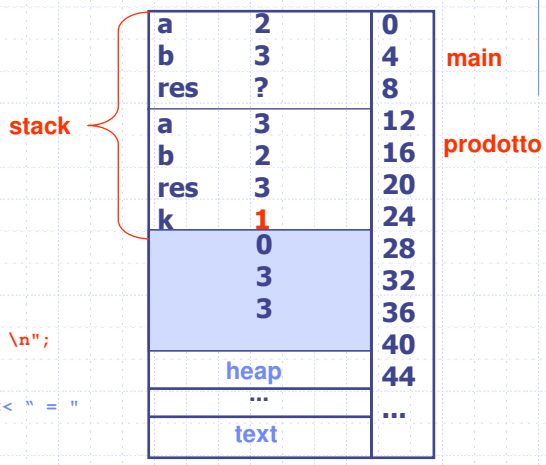


21

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";
}
```

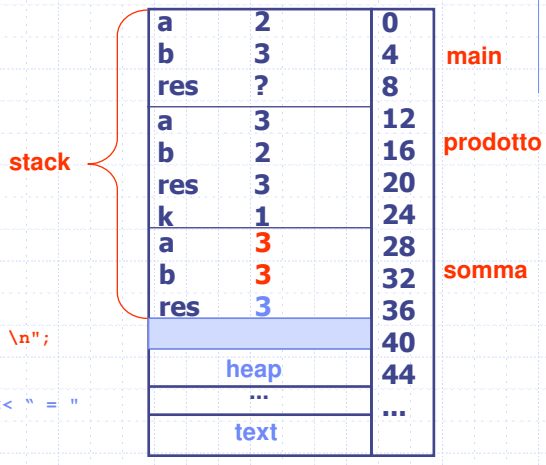


22

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";
}
```

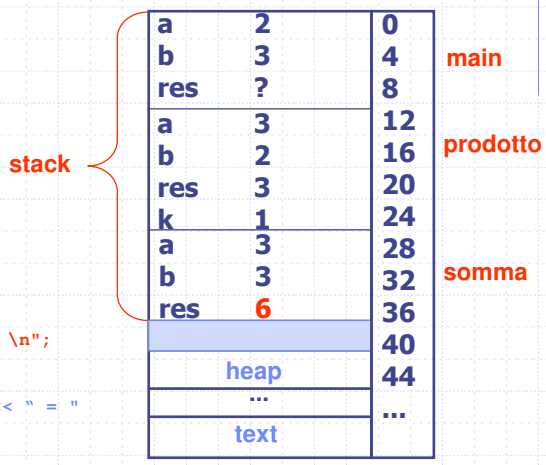


23

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";
}
```

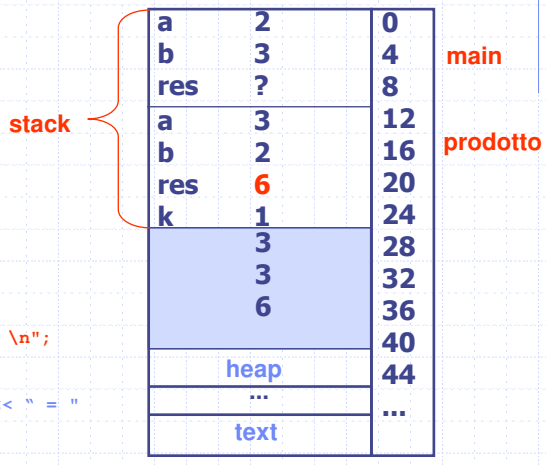


24

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";
}
```

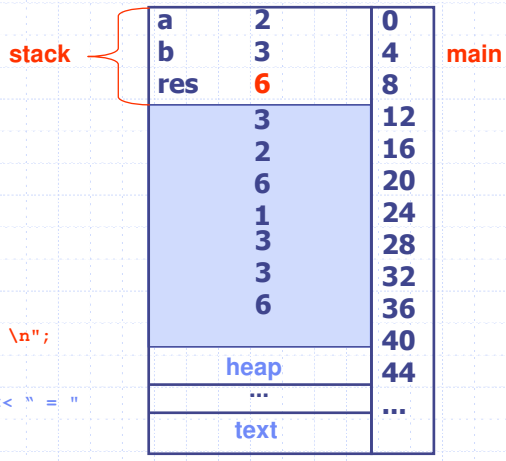


25

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";
}
```



26

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?

27

Funzioni: problema #1

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

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

Quanto vale a quando viene stampata?

I parametri sono passati per valore (copia)!

28

Funzioni: problema #2

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

29

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;
}
```

stack

a	1	0
b	2	4
c	?	8
d	?	12
pa	0	16
pb	4	20
		...

30

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;
}
```

stack

a	1	0
px	0	4
	?	8
	?	12
	?	16
	?	20
		...

OUTPUT: 2

31

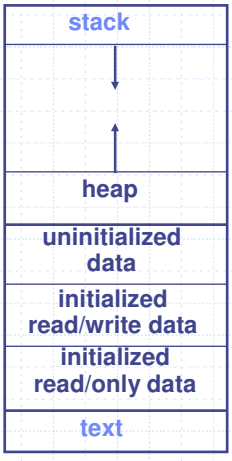
Funzioni e puntatori

TRUCCO: per ottenere più valori di ritorno, passiamo degli indirizzi!

```
void minimax(int a1, int a2, int a3,int *pmin, int *pmax) {
    *pmin=a1; *pmax=a1;
    if (*pmin>a2) *pmin=a2; else if (*pmax<a2) *pmax=a2;
    if (*pmin>a3) *pmin=a3; else if (*pmax<a3) *pmax=a3;
}
main(void) {
   int a,b,c,d,min,max;
   cout << "Dammi 3 numeri\n";
   cin >> a >> b >> c;
   minimax(a,b,c,&min,&max);
   cout << "Il min vale "<<min<<" Il max vale "<<max<<"\n";
}
```

32

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

33

Funzioni, puntatori e reference

VARIANTE: uso delle Reference nel passaggio di parametri a funzione

```
void minimax(int a1, int a2, int a3,int &pmin, int &pmax) {
    pmin=a1; pmax=a1;
    if (pmin>a2) pmin=a2; else if (pmax<a2) pmax=a2;
    if (pmin>a3) pmin=a3; else if (pmax<a3) pmax=a3;
}
main(void) {
   int a,b,c,d,min,max;
   cout << "Dammi 3 numeri\n";
   cin >> a >> b >> c;
   minimax(a,b,c,min,max);
   cout << "Il min vale "<<min<<" Il max vale "<<max<<"\n";
}
```

34

Scope delle variabili

Variabili globali

Nel seguente esempio a e' una variabile globale.
Il suo valore è visibile a tutte le funzioni.
ATTENZIONE! 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
```

35

Scope delle variabili

Variabili 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.

36

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

37

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

38

Principi di Parna

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

39

Scope delle variabili

Variabili statiche "interne"

```
void f(int n) {
    int b;  static int a;
    if (n==0) {a=1 ; b=1;}
    else {a=a+1; b=b+1;}
 cout << "(a,b) in f:"<<a <<" -", <<b<<" - ";
}
void g(void)  {int x=99,y=100,z=101;}
main() {
    int a=5,b=5;
    f(0); g(); f(1);
    cout << "(a,b) in main:"<<a <<" -", <<b<<" - ";
}
```

Output:

```
(a,b) in f: (1,1)  - (a,b) in f: (2,?) - (a,b) in main: (5,5)
```

ATTENZIONE: ? significa che il valore di b alla seconda iterazione NON è (in generale) predicibile!

40

Scope delle variabili

Variabili statiche "esterne"

```
static int a;
void f(int n) {
    ...
}
main() {
    ...
}
```

a è una variabile globale.

Tuttavia è visibile solo alle funzioni che stanno nello stesso file al momento della compilazione!

(Sarebbe un buon meccanismo per implementare le funzioni di libreria, se non fosse che le variabili globali vanno COMUNQUE evitate!)

41

Ancora puntatori e reference

Confronto tra reference e puntatori

```
void f(int* x) { (*x)++; }
void g(int& x) { x++; }
main(void) {
    int a=0;
    f(&a); //ugly, but esplicit
    g(a);  //clean, but hidden
}
```

Attenzione con le reference!

```
int& h() { int x=0; return x;}   // ERROR ! Perché?
int& l() { static int x=0; return x;} //OK, Perché?
```

(esempi presi da B.Eckel, Thinking in C++, pag.452)

42

Tipi di dati

Tipi di dati personalizzati

```
typedef float coordinata;
coordinata z,t;
```

Tipi di dati composti

```
struct punto {
        coordinata x;
        coordinata y;
        }
punto origine;
origine.x=0.0;
origine.y=0.0;
```

43

Puntatori a strutture

Operat. di dereferenziazione di struttura: –>

```
main()
{
    struct point {
      int x;
      int y;
      enum color {BLACK, BLUE, RED, GREEN};
     };
     struct point a, *pa;
     a.x = 3; a.y = 5; a.color=GREEN;
     pa = &a;
     cout<<a.x<<" "<<pa->y <<" "<<(*pa).color;
}
```