Programmazione Avanzata per il Calcolo Scientifico
Lezione N. 5

Luca Formaggia

MOX
Dipartimento di Matematica "F. Brioschi"
Politecnico di Milano

A.A. 2006/2007

Dangling pointers

Introduzione

Queste slides vogliono coprire un aspetto importante, quello del risultato sempre incorretto ma spesso disastroso causato dall'accesso a un puntatore dangling, cioè un puntatore a cui non è associato un indirizzo di memoria valido.

Un primo esempio

```
void dangle(double * a){
double b=5;
a=&b;
}
int main(){
double* pb;
dangle(pb);
cerr«*pb«endl;
}
```

L'esempio compila (senza warnings!), ma il progamma molto probabilmente da `segmentation fault`. (E questo è il caso fortunato: potrebbe semplicemente dare risultati incorretti!) Perchè?

- Primo motivo: la variabile `b` in `dangle` è locale: all'uscita della funzione è distrutta e quindi il suo indirizzo non è valido.
- Secondo motivo: il puntatore a è una varaibile locale di `dangle` a cui viene assegnato (per valore) il contenuto della variabile `pb`, la quale rimane inalterata, e in questo caso non assegnata.

L'esempio compila (senza warnings!), ma il progamma molto probabilmente da `segmentation fault`. (E questo è il caso fortunato: potrebbe semplicemente dare risultati incorretti!) Perchè?

- Primo motivo: la variabile `b` in `dangle` è locale: all'uscita della funzione è distrutta e quindi il suo indirizzo non è valido.
- Secondo motivo: il puntatore a è una varaibile locale di `dangle` a cui viene assegnato (per valore) il contenuto della variabile `pb`, la quale rimane inalterata, e in questo caso non assegnata.

L'esempio compila (senza warnings!), ma il progamma molto probabilmente da `segmentation fault`. (E questo è il caso fortunato: potrebbe semplicemente dare risultati incorretti!) Perchè?

- Primo motivo: la variabile `b` in `dangle` è locale: all'uscita della funzione è distrutta e quindi il suo indirizzo non è valido.
- Secondo motivo: il puntatore a è una varaibile locale di `dangle` a cui viene assegnato (per valore) il contenuto della variabile `pb`, la quale rimane inalterata, e in questo caso non assegnata.

Se `dangle` fosse invece

```
void dangle(double * & a)
static double b;
b=5;
a=&b;
```

non ci sono problemi. La variabile `b` è qui statica e quindi non viene distrutta all'uscita dalla funzione e `a` è passato come referenza.
Tuttavia non riflette un buon stile di programmazione.

I problemi evidenziati negli esempi precedenti sussistono anche se il puntatore fosse passato come argomento di ritorno: il codice seguente è semanticamente incorretto:

```
double * dangle(){
double b=5;
return &b;
}
```

Un esempio più ingannevole

```
void nodangle(double * a)
{a= new double[10];
...
}
int main(){
double* pb;
nodangle(pb);
.... }
```

Un esempio più ingannevole

```
void nodangle(double * a)
{a= new double[10];
...
}
int main(){
double* pb;
nodangle(pb);
.... }
```

Attenzione! a è passata per valore!!. pb rimane inalterato.

```
void nodangle(double * & a)
{a= new double[10];
...
}
```

Ora va bene, ma c'è un'altro problema: la responsabilità di liberare la memoria allocata con il comando `new` è delegata al programma chiamante la funzione. Non è una buona pratica.

```
void nodangle(double * & a)
{a= new double[10];
...
}
```

Ora va bene, ma c'è un'altro problema: la responsabilità di liberare la memoria allocata con il comando `new` è delegata al programma chiamante la funzione. Non è una buona pratica.

```
void nodangle(double * & a)
{...
}
int main(){
double* pb;
pb=new double[10];
nodangle(pb);
....
delete[] pb; }
```

Molto meglio. Ma sarebbe ancora meglio inglobare tutto in una classe e delegare la gestione della memoria al costruttore e distruttore.