Costruzione di Interfacce
Lezione 13
Dal Java al C++

cignoni@iei.pi.cnr.it
http://vcg.iei.pi.cnr.it/~cignoni

Memory Management

- Due grandi categorie di storage:
- Local, memoria valida solo all'interno di un certo scope (e.g. dentro il corpo di una funzione), lo stack;
- Global, memoria valida per tutta l'esecuzione del programma, lo heap.

Local Storage

- {
  int myInteger; // memory for an integer allocated
  // ... myInteger is used here ...

  Bar bar; // memory for instance of class Bar allocated
  // ... bar is used here ...
  }
- '{' e '}' sono i delimitatori di un blocco in c++, Non è detto che corrisponda ad una funzione…
- Non possiamo usare bar, o myInteger fuori dallo scope del blocco!

Global Storage

- Per allocare memoria nel global storage (e.g. per avere puntatori ad oggetti la cui validità persista sempre) si usa l'operatore new.


```
[Bar.H]
class Bar {
public:
    Bar();
    Bar(int a);
    void myFunction(); // this method would be defined
                       // elsewhere (e.g. in Bar.C)
protected:
    int m_a;
};
Bar::Bar(){  m_a = 0;}
Bar::Bar(int a){ m_a = a;}

[main.C]
#include "Bar.H"
int main(int argc, char *argv[])
{ // declare a pointer to Bar; no memory for a Bar instance is
  // allocated now p currently points to garbage
    Bar * p;
    {   // create a new instance of the class Bar (*p)
        // store pointer to this instance in p
        p = new Bar();
        if (p == 0) {
            // memory allocation failed
            return 1;
        }
    }
    // since Bar is in global storage, we can still call methods on it
    // this method call will be successful
    p->myFunction();
}
```


Delete esplicite

- In java si alloca con new e la memoria viene liberata automaticamente dal garbage collector
- In c++ NO. Ci deve pensare l'utente a disallocare esplicitamente quel che ha esplicitanmente allocato con una new
- `delete p; // memory pointed to by p is deallocated`
  Solo oggetti creati con new possono essere deallocati con delete.

Memory Management

```
[Foo.H]
#include "Bar.H"

class Foo {
private:
    Bar* m_barPtr;
public:
    Foo() {}
    ~Foo() {}
    void funcA() {
        m_barPtr = new Bar;
    }

    void funcB() {
        // use object *m_barPtr
    }

    void funcC() {
        // ...
        delete m_barPtr;
    }
};
```


Memory Management

```
{
    Foo myFoo; // create local instance of Foo
    myFoo.funcA(); // memory for *m_barPtr is allocated

    // ...
    myFoo.funcB();
    // ...
    myFoo.funcB();
    // ...

    myFoo.funcC(); // memory for *m_barPtr is deallocated
}
```


Memory Management

```
{
    Foo myFoo;
    //...
    myFoo.funcB(); // oops, bus error in funcB()

    myFoo.funcA(); // memory for *m_barPtr is allocated

    myFoo.funcA(); // memory leak, you lose track of the memory previously
                   // pointed to by m_barPtr when new instance stored
    //...
    myFoo.funcB();

}   // memory leak! memory pointed to by m_barPtr in myFoo
    // is never deallocated
```


Costruttori e Distruttori

- La soluzione corretta è quella di delegare il compito di allocare e disallocare ai costruttori e distruttori degli oggetti


```
[Foo.H]
#include "Bar.H"

class Foo {
private:
    Bar* m_barPtr;
public:
    Foo();
    ~Foo();
    void funcA() {  ...   }

    void funcB() {
       // use object *m_barPtr
    }
    void funcC() {  ...    }
};

Foo::Foo(){   m_barPtr = new Bar; }
Foo::~Foo(){  delete m_barPtr; }
```


Istanze, puntatori e riferimenti

- Piccolo riepilogo
  - L'allocazione, disallocazione della memoria delle istanze degli oggetti è gestita automaticamente. Quando una variabile esce dal proprio scope la sua mem è disallocata (anche se c'era un puntatore che la riferiva)
  - I puntatori ottenuti tramite new sono relativi a porzioni di memoria di l'utente ha la responsabilità
  - I riferimenti sono semplicemente nomi differenti con cui si riferisce altre variabili/oggetti, quindi non si può/deve gestirne direttamente la memoria (ma si deve gestire la mem della variabile cui si riferiscono).

Parametri

- In java il passaggio di parametri è sempre per riferimento, cioè si può cambiare un oggetto usandolo come argomento di una funzione
- In c++ i parametri possono essere passati per riferimento o per valore (default)

```
void IncrementByTwo(int foo) { foo += 2; } // foo non cambia
void IncrementByTwo(int &foo) { foo += 2; } // foo cambia
```

- Alternativamente, si può ottenere lo stesso effetto passando (per valore) un puntatore
- `void IncrementByTwo(int* fooPtr) { *fooPtr += 2; }`

Valori di ritorno

- I valori di ritorno di funzione possono essere, al solito, istanze puntatori o riferimenti
- Errore frequente, restituire riferimenti o puntatori a istanze locali alla funzione
- Tali istanze sono distrutte al ritorno dalla funzione, quindi si ha puntatori o riferimenti a cose non più allocate

Errore return 1

```
[FooFactory.C]
#include "FooFactory.H"
#include "Foo.H"

Foo* FooFactory::createBadFoo(int a, int b) {

    Foo LocInst (a,b); // creates an local instance of class Foo
    return & LocInst;  // returns a pointer to this instance

} // ERROR! LocInst leaves scope
  // and it is destroyed!
```


Errore ritorno 2

```
Foo& FooFactory::createBadFoo(int a, int b) {

    Foo aLocalFooInstance(a,b); // creates an local instance
                                // of the class Foo
    return aLocalFooInstance;   // returns a reference to this
                                // instance

} // EEK! aLocalFooInstance leaves scope and is destroyed!
```


Soluzione corretta

```
Foo* FooFactory::createFoo(int a, int b) {
    return new Foo(a,b);  // returns a pointer to an
instance of Foo
}

Foo FooFactory::createFoo(int a, int b) {
    return Foo(a,b);  // returns an instance of Foo
}
```

morale:

MAI ritornare puntatori a oggetti che non sono stati generati con new, a meno che non si sia estremamente sicuri che le variabili cui si riferiscano non escano dallo scope

Per chi viene dal C…

- Differenze tra new e malloc
- Malloc non conosce per che cosa serve la memoria e quindi è l'utente a dover fare I conti di quanta ne serve
- Malloc non inizializza (calloc inizializza solo a valori costanti),new chiama il costruttore per ogni oggetto allocato
- Similarmente delete chiama il distruttore per ogni oggetto disallocato
- MAI mescolare new e malloc…

New e delete di array

- Obj *op = new Obj[20]; // allocates 20 Obj
- Per ogni oggetto allocato viene chiamato il costruttore
- Notare che la delete deve essere fatta con la delete per array
- delete [] op;

Esempio

```
class Person
    {
        public:
            Person()
            {}
            Person(char const *n, char const *a,
                   char const *p);
            ~Person();

            char const *getName() const;
            char const *getAddress() const;
            char const *getPhone() const;

        private:
            // data fields
            char *name;
            char *address;
            char *phone;
    };
```


Class Person

Scopo del costruttore è inizializzare i campi dell'oggetto

```
#include "person.h"
    #include <string.h>

    Person::Person(char const *n, char const *a, char
const *p)
    {
        name    = strdupnew(n);
        address = strdupnew(a);
        phone   = strdupnew(p);
    }
```


Strdup con new

Dove

```
char *strdupnew(char const *str)
    {
        return str ? strcpy(new char [strlen(str) + 1], str) :
0;
    }
```

questo per essere sicuri di non mescolare malloc e new… (strdup usa malloc!)


```
#include "person.h"
 #include <string.h>

 Person::~Person()
 {
     delete name;
     delete address;
     delete phone;
 }
```


Tipico uso

```
#include "person.h"
#include <iostream>

void showPerson()
{
    Person
        karel("Karel", "sdfdfdee", "038 420 1971"),
        *frank = new Person("Frank", "Oostu", "050 403 2223");

    cout << karel.getName()     << ", " <<
            karel.getAddress()  << ", " <<
            karel.getPhone()    << endl <<
            frank->getName()    << ", " <<
            frank->getAddress() << ", " <<
            frank->getPhone()   << endl;

    delete frank;
}
```

Esempio2

```
Person::~Person(){cout <<"Person destructor called"<< endl;}

int main()
 {
     Person *a  = new Person[2];
     cout << "Destruction with []'s" << endl;
     delete [] a;
     a = new Person[2];
     cout << "Destruction without []'s" << endl;
     delete a;
     return 0;
 }
 Generated output:
Destruction with []'s
Person destructor called
Person destructor called
Destruction without []'s
Person destructor called
*/
```


- Il precedente esempio generava memory leaks, solo il distruttore del primo elemento viene chiamato se si usa la sintassi sbagliata della delete

Esempio3

```
Person::~Person(){cout <<"Person destructor called"<< endl;}

int main()
{
    Person   **a;

    a = new Person* [2];

    a[0] = new Person [2];
    a[1] = new Person [2];

    delete [] a;

    return 0;
}
No Output!!
```


- Nessun output perché 'a' è un array di puntatori. La disallocazione di un puntatore (e quindi di NON di un oggetto) non comporta alcuna azione.
- Ogni elemento dell'array andrebbe disallocato singolarmente.

Operatore assegnamento

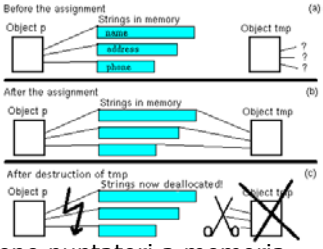
- L'operatore di assegnamento di default in C++ fa la copia byte a byte dei campi delle due strutture.
- Se si usa puntatori questo è assai pericoloso

Operatore Assegnamento

```
void printperson(Person const &p)
    {
        Person
            tmp;

        tmp = p;
        cout << "Name:    "<< tmp.getName()    << endl <<
                "Address: "<< tmp.getAddress()<< endl <<
                "Phone:   "<< tmp.getPhone()  << endl;
    }
```

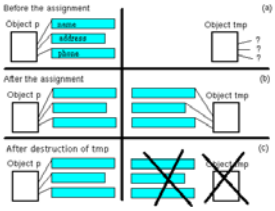
Operatore Assegnamento



- Alla fine p contiene puntatori a memoria disallocata!

Operatore Assegnamento

- Il problema è dovuto al fatto che l'operatore assegnamento ha fatto la copy bytewise dei puntatori ignorando il loro significato
- Approccio giusto:



Overloading Operatore =

- In c++ si può fare overload dei vari operatori
  - Sintassi
  - Basta definire una funzione chiamata operator=(…)
  - Si può fare per I vari operatori
    - operator+()
    - Ecc.

Operator =

```
Person &Person::operator=(Person const &other)
{
    if (this != &other)
    {
        delete address;
        delete name;
        delete phone;

        address = strdupnew(other.address);
        name = strdupnew(other.name);
        phone = strdupnew(other.phone);
    }
    // return current object. The compiler will
    // make sure that a reference is returned
    return *this;
}
```


Note operator=

- a=b è equivalente a a.operator=(b);
- if(this!=other) serve per evitare l'autoassegnamento (altrimenti facendo p=p si distruggerebbe le stringhe prima di poterle copiare)
- Il valore di ritorno serve per l'assegnamento multiplo a=b=c; (in C++ anche l'assegnamento è un expr.)

Costruttore di copia

- Costruttori che hanno in ingresso un riferimento ad un oggetto della stessa classe
- Serve per inizializzare una variabile con un'altra

```
class Person
    {
        public:
            Person(Person const &other);
    };
```

Costruttori di copia e assegnamento

- Entrambi devono duplicare un oggetto
- Il costruttore di copia non deve disallocare memoria (l'oggetto è stato appena creato)
- Il costruttore di copia non deve fare controlli di autoassegnamento (non si può inizializzare una var con se stessa)

Costruttori di copia e parametri

```
void nameOf(Person p)    // no pointer, no reference
    {                    // but the Person itself
        cout << p.getName() << endl;
    }
```

- il costruttore di copia è chiamato quando si passa un oggetto per valore

Costruttore di copia e return value

```
Person getPerson()
{
 string
      name,
      address,
      phone;

 cin >> name >> address >> phone;

 Person
     p(name.c_str(), address.c_str(),phone.c_str());

 return p;           // returns a copy of `p'.
}
```

- il costruttore di copia è chiamato quando una funzione restituisce un'istanza di un oggetto.

Const e parametri

- Quando modifica una dichiarazione di dati, const specifica che l'oggetto o la variabile non è modificabile
- Quando segue la lista dei parametri di una funzione membro, const specifica che la funzione non modifica l'oggetto per cui è invocata.

```
const int i = 5;

i = 10; // Error
i++;    // Error
```


Const e puntatori

```
char *const aptr = mybuf;  // Constant pointer
// non posso cambiare il valore (indirizzo)
// del puntatore
*aptr = 'a';       // Legal
aptr = yourbuf;    // Error
```

- un puntatore ad una variabile const può essere assegnato solo ad un puntatore che è dichiarato const

```
const char *bptr = mybuf;// Pointer to constant data
// non posso cambiare il contenuto della locazione
// puntata.
*bptr = 'a';       // Error
bptr = yourbuf;    // Legal
```

- può essere usato per impedire ad una funzione di cambiare un parametro passato per puntatore
- **Trucco leggere da destra a sinistra**

Const e riferimenti

- Un uso importante dei riferimenti costanti è il loro uso nel passaggio di parametri
- void printperson (Person const &p)
- Si evita di invocare il costruttore di copia, pur rimanendo sicuri che non modifica l'oggetto

Morale

- Se la nostra classe contiene puntatori conviene sempre scrivere accuratamente
  - Costruttore
  - Costruttore di copia
  - Operatore di assegnamento
  - Distruttore

Template

- I Template sono un meccanismo che permette di definire funzioni e classi basate su argomenti e oggetti dal tipo non specificato

```
template class <T> swap(T &a, T &b);

template class <T> class List {...};
```

- Queste funzioni e oggetti generici diventano codice completo una volta che le loro definizioni sono usate con oggetti reali

Esempio di template di funzione

Scambio tra due oggetti generici:

```
template <class T> void swap(T &a, T &b){
   T tmp = a;
   a = b;
   b = tmp;
}

int main(){
   int a = 3, b = 16;
   double d = 3.14, e = 2.17;
   swap(a, b);
   swap(d, e);
   // swap (d, a); errore in compilazione!

   return (0);
}
```


Template Osservazioni

- Notare che la definizione di una funzione template è simile ad una macro, nel senso che la funzione template *non è ancora codice*, ma lo diventerà una volta che essa viene usata
- Il fatto che il compilatore generi codice concreto solo una volta che una funzione è usata ha come conseguenza che una template function non può essere mai raccolta in una libreria a run time
- Un template dovrebbe essere considerato come una sorta di dichiarazione e fornito in un file da includere.