costruttori e distruttori

Costruttore

- E’ un metodo che ha lo stesso nome della classe a cui appartiene: serve per inizializzare un oggetto all’atto della sua creazione
- Ce ne possono essere uno, più di uno, nessuno
- Può avere un numero qualunque di argomenti, ma nessun valore di ritorno
- Non viene chiamato direttamente. Viene eseguito automaticamente alla creazione di un oggetto

Costruttore

```
class Rettangolo {

  private:
      int base;
      int altezza;
  public:
      Rettangolo(int,int);
    ...
};
```

Dichiariazione di un costruttore

```
Rettangolo::Rettangolo(int b,int a) {
  base=b; altezza=a; }
```

Definizione del costruttore

Costruttore

```
#include <Rettangolo.h>

....
void main() {
  Rettangolo r(10,20);

  Rettangolo *rp=new Rettangolo(20,30);
...
}
```

La creazione di un oggetto Rettangolo ha ora bisogno di due parametri che inizializzano gli attributi interni del nuovo oggetto

Distruttore

- E’ un metodo avente il nome della classe a cui appartiene preceduto da ~
- Ce ne può essere al massimo uno
- Non viene chiamato direttamente. È eseguito automaticamente quando viene liberata una area di memoria che contiene un oggetto della classe
- Non può accettare parametri né avere valori di ritorno

Distruttore

```
class Rettangolo {
  public:
      ~Rettangolo();
   ...
};
```

Dichiarazione del distruttore

```
Rettangolo::~Rettangolo() {
  cout << “Addio mondo crudele...”; }
```

Definizione del distruttore

Constructor1.cpp

Scope & Initialization

- Si può dichiarare una variabile in qualunque parte di un blocco (non solo all'inizio)
- Il costruttore viene attivato in corrispondenza della dichiarazione
- Il distruttore viene attivato quando l'esecuzione lascia l'ambito di validità della variabile (out of scope)

Initialization & Scope

- Indicazione di stile:
  Ridurre al minimo necessario lo scope (dichiarazione al primo uso, es. for)

- Non si può inserire una definizione di variabile con costruttore in una parte di codice eseguita sotto condizione.

Allocazione statica

- In C++ l'allocazione statica per le variabili viene effettuata quando si incontra la dichiarazione della variabile. Quando il compilatore trova:

```
Poligono triangolo;
```

provvede a inserire nel codice oggetto le istruzioni per riservare lo spazio di memoria relativo

Allocazione dinamica

- L’allocazione dinamica viene effettuata tramite l’operatore *new* che alloca una nuova area di memoria durante l’esecuzione.
- L’area di memoria allocata dinamicamente non ha un nome associato, ma viene utilizzata tramite puntatori
- La memoria deve essere rilasciata esplicitamente quando non serve più tramite l’operatore *delete*

Allocazione dinamica

- Nel caso di oggetti allocati dinamicamente:

```
Poligono* pPol;
pPol=new Poligono;
```

alloca un oggetto dinamicamente. Notare che la notazione per mandare messaggi ad un oggetto tramite un puntatore ad esso è uguale a quella per accedere ai campi di una struttura tramite puntatore: si usa *l'operatore ->*

Allocazione dinamica

```
class Stringa{
      char * str;
      public:
            Stringa(int );
            ~Stringa();

Stringa::Stringa(int size) {
      str = new char[size+1];
      *str = '\0';}
Stringa::~Stringa() {
      delete [] str;}
```

creazione del vettore di char e inizializ. alla stringa vuota

esempio: Stash

Stash3.h

```
class Stash {
  ...
public:
  Stash(int size);
  ~Stash();
  ...
```

Sostituiscono Initialize() e Cleanup()

Stash3.cpp

```
Stash::Stash(int sz) {
  size = sz;
  quantity = 0;
  storage = 0;
  next = 0;
}
```

```
Stash::~Stash() {
  if(storage != 0) {
    cout << "free ...";
    delete []storage;
  }
```

esempio: Stash

```
int main() {
  Stash intStash;
  intStash.initialize(sizeof(int));
  ...
  Stash stringStash;
  const int bufsize = 80;
  stringStash.initialize(sizeof(char) * bufsize);
  ...
  intStash.cleanup();
  stringStash.cleanup();
}
```

Stash3Test.cpp

```
int main() {
  Stash intStash(sizeof(int));
  ...
  Stash stringStash(sizeof(char) * bufsize);
  ...
}
```

esempio: Stack

```
class Stack {
  struct Link {
    Link(void* dat, Link* nxt);
    ~Link();
  }
...
public:
  Stack();
  ~Stack();
  ...
```

Stack2.h

Stack2.cpp

```
Stack::Stack() {
  head = 0; }
Stack::~Stack() {
  require(head == 0,
    "Stack not empty");
}
```

```
Stack::Link::Link(void*
      dat, Link* nxt) {
  data = dat;
  next = nxt;
}
Stack::Link::~Link(){ }
```

```
void Stack::push(void* dat){
  head = new Link(dat,head);
}
```

Inizializzazione di aggregati

- Facilitazione sintattica per array & co.

```
int a[3] = { 1, 2, 3 };
```

ma anche:

```
int a[3] = {0};
Aclass X[] = { {1,2,'a'}, {3,4,'b'} };
```

se c'è un costruttore ...

```
Bcls X[] = {Bcls(1,'a') Bcls(2,'b')}
```

- Conteggio automatico:

```
int a[] = { 1, 2, 3 };
quanti = sizeof a / sizeof *a ;
```

Multiarg.cpp

- *... ma non sarà meglio essere espliciti ? ...*

Quali nomi ?

- Name decoration: il completamento automatico dei nomi fatto dal compilatore per evitare conflitti:
  - nome globale:
    ```
    void f();   --> _f
    ```
  - nome locale:
    ```
    class X { void f(); };   --> _X_f
    ```

Nomi completi

- In C++ la name decoration comprende anche i tipi degli argomenti:
  - nome globale:
    ```
    void f(int);   --> _f_int
    ```
  - nome locale:
    ```
    int X::f(int){ };   --> _X_f_int
    ```
  - nome sovradefinito (overloaded):
    ```
    int X::f(float){ }; --> _X_f_float
    ```

Overloading di costruttori

- L'overloading è particolarmente utile per i costruttori - più modi di inizializzare:
  - Senza argomenti:

    ```
    AClass::AClass(){ a=0; x=0; };
    ```

  - Con argomenti:

    ```
    AClass::AClass(int A, int X){ a=A; x=X; };
    AClass::AClass(int A){ a=A; x=0; };
    ```

  - il contesto decide quale si usa:

    ```
    AClass Prova;
    AClass Test(1);
    AClass Pippo(1,2);
    ```

esempio: Stash

Stash3.h

```
class Stash {
public:
  Stash(int size);
  Stash(int size, int initQuantity);
  ~Stash();
  ...
```

Inizializza e crea elementi

Stash3.cpp

```
Stash::Stash(int sz, int initQuantity) {
  size = sz;
  quantity = 0;
  storage = 0;
  next = 0;
  inflate(initQuantity);
}
```

```
int main() {
  Stash intStash(sizeof(int));
  ...
  Stash stringStash(sizeof(char)
        * bufsize, 100);
  ...
}
```

Stash3Test.cpp

Argomenti di default

- Il codice dei due costruttori è quasi uguale ...
- Funzioni con argomenti di default (dichiaraz.)

```
int UnaFunzione(int a, int b = 0);
Stash(int size, int initQuantity = 0);
```

- Regola per gli argomenti opzionali:
  - solo gli ultimi

```
int F1(int a, int b=0, int c=0, int d=0);
int F1(int a=0, int b, int c, int d);
int F1(int a, int b=0, int c, int d=0);
```

Un solo costruttore !

Overloading o Default ?

- La scelta dipende da:
  - quanto si differenziano le funzioni
  - quanto si vogliono evidenziare usi diversi
- In alcuni casi l'inizializzazione con valori di default non è la stessa cosa dell'inizializzazione senza argomenti

esempio: Mem

```
class Mem {
  byte* mem;
  int size;
  void ensureMinSize(int minSize);
public:
  Mem();
  Mem(int sz);
  ~Mem();
  int msize();
  byte* pointer();
  byte* pointer(int minSize);
}
```

```
Mem::Mem() { mem = 0; size = 0; }
Mem::Mem(int sz) {
  mem = 0;
  size = 0;
  ensureMinSize(sz);
}
```

```
byte* Mem::pointer() { return mem; }
byte* Mem::pointer(int minSize) {
  ensureMinSize(minSize);
  return mem;
}
```

```
class Mem {
public:
  Mem(int sz=0);
  byte* pointer(int minSize=0);
}
```

esempio: Mem

```
class MyString {
  Mem* buf;
public:
  MyString();
  MyString(char* str);
  ~MyString();
  void concat(char* str);
  void print(ostream& os);
};
```

```
MyString::MyString() {  buf = 0; }
MyString::MyString(char* str) {
  buf = new Mem(strlen(str) + 1);
  strcpy((char*)buf->pointer(),str);}
```

... ne vale la pena ?

```
MyString(char* str=0);
```

```
MyString::MyString(char* str) {
  if (!*str) {
    buf=0; return;
  }
  buf = new Mem(strlen(str) + 1);
  strcpy((char*)buf->pointer(),str);
}
```

esercizio (7.1)

- Creare una classe **Text** che contenga un oggetto String in cui memorizzare il contenuto di un file di testo.
  Definire due costruttori: un costruttore di default e uno che prenda come argomento una String (nome del file da leggere).
  Il secondo costruttore apre il file e ne legge il contenuto nella variabile privata String.
  Aggiungere un metodo **contents()** che ritorna il contenuto della variabile privata.
  Da main, creare un oggetto definendo un oggetto di classe Text e visualizzare il contenuto del file letto.